APL2 Programming:

Processor Interface Reference

Version 2 Release 1

SH21-1058-00

APL2 Programming:

Processor Interface Reference

Version 2 Release 1

SH21-1058-00

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

First Edition (March 1992)

This edition applies to Release 1 of APL2 Version 2, Program Number 5688-228, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest edition of the applicable IBM system bibliography for current information on this product.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A Reader's Comment Form is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, P.O. Box 49023, Programming Publishing, San Jose, California, U.S.A. 95161-9023. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Programming Interface Information

This book is intended to help you use the auxiliary processors, associated processor function routines, and the services available to those processors and routines provided by APL2 Version 2. This book documents General-Use Programming Interface and Associated Guidance Information provided by APL2 Version 2.

General-Use programming interfaces allow the customer to write programs that obtain the services of APL2 Version 2.

Trademarks and Service Marks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

APL2

ESA

GDDM

IBM

MVS/XA

S/390

System/370

System/390

About This Book

This manual describes the interfaces to auxiliary processors and associated processor external routines, the services available to those processors and routines, the interfaces for calling APL2* from non-APL programs, and the services available to those programs.

IBM provides a set of auxiliary processors described in *APL2 Programming: System Services Reference*, and a set of associated processor external routines described in *APL2 Programming: Using the Supplied Routines*. Those manuals should be consulted for use of the IBM facilities. The present manual provides reference material for programmers who intend to write their own processors or external routines.

This manual also provides material for programmers who intend to call APL2 routines from programs written in other languages. You can find other material about this topic in the discussion of the *APL2PIE* function in *APL2 Programming: Using the Supplied Routines*.

Most of the interfaces described in this book are available under both CMS and TSO. Differences in availability and in use do exist, however, and are noted throughout the manual.

Auxiliary processors written for VS APL can, in most cases, continue to be used with APL2. This manual describes the limitations in that compatibility, as well as additional facilities available to such processors, but does not define interfaces to VS APL. See *VS APL for CMS and TSO: Writing Auxiliary Processors* for that information.

Audience

The portions of this manual that discuss auxiliary processors and associated processor external routines assume that you are an experienced programmer in System/370* or System/390* assembler language. Although the interfaces defined here can be used from several compilable languages, they are described using assembler language terminology.

The portions of this manual that discuss calling APL2 from non-APL programs assume that you are familiar with other programming languages. The interfaces are described using assembler language terminology and examples are provided in several different languages.

The manual also makes no attempt to describe the facilities of the APL language, its system environment, or the operating systems under which it executes. The manuals listed at the end of this preface are sources for this information.

Organization

This manual is organized in four parts:

- “Part One: Introduction to Processors and External Routines” on page 1 explains a number of concepts used throughout the manual.
- “Part Two: Interfaces and Services from APL2” on page 21 describes the set of interfaces and services that are available under APL2.

- "Part Three: Using VS APL Processors under APL2" on page 113 discusses changes you may need or want to make to VS APL auxiliary processors when migrating them to an APL2 environment.
- "Part Four: Calls to APL2 from Non-APL Programs" on page 117 discusses a facility in which a non-APL program can start APL2, call APL2 routines, and terminate APL2.

APL2 Publications

The books in the APL2 library are shown in Figure 1. This figure shows which books can help you with specific tasks such as finding reference information.

Figure 1. APL2 Licensed Program Library

Task	APL2 Publication	Publication Number
Evaluating APL2	*APL2 General Information*	GH21-1051
	APL2 Application Environment Licensed Program Specifications	GH21-1063
	APL2 Licensed Program Specifications	GH21-1070
Installing APL2	*APL2 Installation and Customization under CMS*	SH21-1062
	APL2 Installation and Customization under TSO	SH21-1055
Migrating to Version 2 Release 1	*APL2 Migration Guide*	SH21-1069
Finding Reference Information	*APL2 Programming: Language Reference*	SH21-1061
	APL2 Programming: Processor Interface Reference	SH21-1058
	APL2 Programming: System Services Reference	SH21-1054
	APL2 Reference Card	SH21-1071
	APL2 Reference Summary	SX26-3999
Programming	*An Introduction to APL2*	SH21-1073
	APL2 GRAPHPAK: User's Guide and Reference	SH21-1074
	APL2 Programming: Guide	SH21-1072
	APL2 Programming: Using Structured Query Language (SQL)	SH21-1057
	APL2 Programming: Using the Supplied Routines	SH21-1056
Diagnosing Problems	*APL2 Diagnosis*	LY27-9601
	APL2 Messages and Codes	SH21-1059

Related Publications

VS APL Publications

- *VS APL for CMS and TSO: Writing Auxiliary Processors,* SH-20-9068

VM/CMS Publications

- *Virtual Machine/System Product: CMS Command Reference,* SC19-6209
- *Virtual Machine/System Product: CMS User's Guide,* SC19-6210
- *Virtual Machine/System Product: CP Command Reference for General Users,* SC19-6211
- *Virtual Machine/System Product: System Programmer's Guide,* SC19-6203

MVS Publications

- *OS/VS Virtual Storage Access Method (VSAM) Options for Advanced Applications,* GC26-3819
- *OS/VS Virtual Storage Access Method (VSAM) Programming Guide,* GC26-3838
- *OS/VS2 System Programming Library: Data Management,* GC26-3830
- *OS/VS2 MVS Data Management Macro Instructions,* GC26-3873
- *OS/VS2 MVS Data Management Services Guide,* GC26-3875
- *OS/VS2 MVS Supervisor Services and Macro Instructions* GC28-0683
- *OS/VS2 TSO Guide to Writing a Terminal Monitor Program or Command Processor,* GC28-0648
- *OS/VS2 TSO Terminal User's Guide,* GC28-0645

MVS/XA Publications

- *MVS/Extended Architecture Data Management Macro Instructions,* GC26-4014
- *MVS/Extended Architecture Supervisor Services and Macro Instructions,* GC28-1154
- *MVS/Extended Architecture System Programming Library: Data Management,* GC26-4010
- *MVS/Extended Architecture System Programming Library: 31-Bit Addressing,* GC28-1158
- *MVS/Extended Architecture TSO/E Guide to Writing a Terminal Monitor Program or Command Processor,* SC28-1136
- *MVS/Extended Architecture VSAM Reference,* GC26-4016

Part One: Introduction to Processors and External Routines

Chapter 1. Auxiliary Processor Concepts

Introduction

In theory, an auxiliary processor is an asynchronous program that is communicating with an APL2 session. In practice, it is usually providing a service to that session. Obvious services are to read data from a file or write data to a file. However, the services that can be provided are limited only by the imagination of the designer and the facilities available in the system under which the auxiliary processor is executing.

Auxiliary processors use the *Shared Variable Processor* (SVP) to communicate with an APL2 session. The SVP is a component of APL2 that provides a communication path between two independent processors. The two processors may be separate tasks within a single MVS address space, separate pseudo-tasks within a CMS virtual machine, or tasks running in two different MVS address spaces or CMS virtual machines or processes running on two different systems connected with TCP/IP.

It is also possible for two auxiliary processors to communicate with each other using the SVP. In reality this is what always happens, since the APL2 interpreter is acting as an auxiliary processor on behalf of the APL user session.

Auxiliary processors exchange information with each other through *shared variables*. A variable may not be shared by more than two auxiliary processors. If a processor needs to communicate with more than one other processor, it must share a different variable with each of them. A variable becomes fully shared when each processor has offered to share it with the other. The two processors are then called *share partners*. A number of variables may be shared between two share partners.

Note that either processor may make the initial offer. Until a *counter offer* has been made, the offer is *outstanding*, and only the one processor views the variable as shared. Each processor is notified when the other offers a variable.

APL sessions share variables and obtain information about share offers by using the $\square SVO$ and $\square SVQ$ system functions. These functions are described in *APL2 Programming: Language Reference*. In its discussion of shared variables and auxiliary processors, *APL2 Programming: System Services Reference* explains the use of $\square SVO$ and $\square SVQ$ in communicating with auxiliary processors.

A shared variable normally has a value, though it may initially be shared without any value. Each partner may **set** a value for the shared variable, and may **use** (that is, look at) its current value. The SVP provides synchronization to ensure that the share partners have a consistent view of the variable's value.

- If both share partners have been informed of the current value, the SVP does not retain it, since each partner is assumed to have a copy of it.
- When one partner **sets** the variable, that partner is assumed to retain a copy of it, but the SVP also maintains a copy in *shared memory*.

A processor's copy may become out of date almost immediately after a **set** or **use**. The processor may always ask for an updated copy of the value. The SVP will

either return the current value, or will indicate that the processor already has the correct value.

Share partners normally establish *protocols* governing the order in which variables are **set** and **used**. The SVP assists in this process by returning information on the *access state* of a variable, and by maintaining an *access control vector* for each variable. Both share partners may inspect and modify the access control vector.

Either share partner may *retract* a variable at any time. When this occurs, the variable becomes an outstanding offer. When the other partner retracts the variable, it ceases to be shared. A processor is notified when its share partner retracts a fully shared variable.

Scheduling Processors

The rules for starting and dispatching auxiliary processors depend on whether they are *global* or *local*, and under which operating system they execute. These rules, in turn, determine some of the information you need to know to write an auxiliary processor.

Local auxiliary processors: These processors share their execution environment with an APL2 session, so are started when a user invokes APL2. They are identified by the **APNAMES** invocation option, or the **ATASKS** or **RESAPS** installation options. (The installation options are used only if the processor is to be physically combined with the APL2 product. In this case, the **ATASKS** option is used for processors written to the interface defined in Chapter 4, “Entry and Exit Conditions” on page 22, while the **RESAPS** option is used for processors written to the VS APL interface.)

Local processors have read/write access to the same storage used by APL2. This has the disadvantage that an error in your processor could destroy a user's APL2 session. It has the advantage that you can program the processor to make use of a wide range of services that are available to APL2 itself. These include data conversions, an APL file system, message formatting, and terminal sharing.

Global auxiliary processors: These auxiliary processors have their own execution environment. This means they can freely use any facilities provided by the operating system without any chance of interfering with APL2 or other auxiliary processors. (In VM there is one restriction: The processor should not use VMCF.)

One important distinction is that global processors are normally shared by many APL2 sessions. Consider, for example, two users sharing a variable with processor 333. If 333 is a *local* auxiliary processor, each APL2 session will have a separate "instance" of it, and the logic within the processor will be aware of only one partner. If 333 is a *global* auxiliary processor, there is only one "instance" of the processor in the system, and it must normally be prepared to share variables concurrently with multiple APL2 sessions. The good part of this is that global processors are well suited for global resource managers. For example, an installation could write a global processor that provided updates to system files. The processor could include authorization checks, and also resolve concurrent update problems.

Because global processors are not associated with a particular APL2 session, they have no access to an individual user's terminal. They should be thought of as being started by (or on behalf of) the system operator, and should provide for communication with the system console.

With a VM operating system: *Local* auxiliary processors share a CMS virtual machine with other similar auxiliary processors and with the APL2 session that started them. This is possible because the APL2 session provides multitasking, including services to *signal* (or post) and to *wait* for signals. Those services are described in Chapter 10, "Code P__: Process Services" on page 65. APL2 also provides compatible support for the **ASVPWAIT** service provided in VS APL.

Global auxiliary processors execute in their own CMS virtual machine. This is normally a disconnected machine that is started by a CP **AUTOLOG** command.

With an MVS or MVS/XA operating system: *Local* auxiliary processors share an MVS address space with other similar auxiliary processors and with the APL2 session that started them. (This is normally a TSO session, but may be a batch job that is executing APL2 under **PGM=IKJEFT01**.) Each auxiliary processor executes as a separate MVS task and can use MVS tasking facilities, such as **WAIT** and **POST**. Local processors may also choose to use the services described in Chapter 10, "Code P__: Process Services" on page 65. These make it easier to produce auxiliary processors that can be used in both CMS and TSO environments.

Global auxiliary processors execute in their own MVS address space. They may execute either as an MVS batch job or a started job.

Global and Local Shared Variable Processors

APL2 implements the SVP as a two-level hierarchy. The two levels are the *local SVP* and the *global SVP*. Each level of the SVP uses a data area referred to as *shared memory*, so there is also local shared memory and global shared memory.

The local SVP is established in each user's address space (MVS) or virtual machine (VM) when APL2 is invoked, and allocates local shared memory within the address space or machine.

The global SVP is established, typically by a system operator, as a separate address space or virtual machine. Global shared memory is allocated in CSA for MVS or in a writable saved segment for VM. The global SVP is an optional component of the APL2 system and need not be installed or invoked by an installation.

The APL2 interpreter interfaces with both the local and global SVP, so it can share variables with local auxiliary processors, global auxiliary processors, and other APL2 users. If the global SVP is not active, or does not authorize the APL2 user when APL2 is invoked, the APL2 user will only be able to share variables via local auxiliary processors.

APL2 cooperative processing through TCP/IP depends only on a local auxiliary processor, not the global SVP, so in some cases it is possible to share variables between sessions without global SVP authorization.

If the global SVP is available and the APL2 user is authorized to use it, the following rules apply:

- Shared variable offers from an APL2 user are processed first by the local SVP. If an active local auxiliary processor is found with the specified processor id, the offer will be made to that processor. If the processor id specified does not match that of any of the active local auxiliary processors, the offer will be extended in global shared memory.
- General offers (processor id 0) are extended only in global shared memory.
- Shared variable queries (⎕SVQ) obtain information from both the local SVP and the global SVP if it is available.
- If a local auxiliary processor and a global auxiliary processor are using the same processor number, the APL2 user is able to share variables only with the local auxiliary processor.

Processor Identification

An auxiliary processor is conceptually identified by an 8-byte token. In practice a fullword integer is normally used, since the APL language supports only such integers when sharing with auxiliary processors. The processor number is expressed as a binary integer and placed in the first four bytes of the field, with binary zeros in the last four bytes.

Auxiliary processor numbers must be distinct from APL user numbers. By convention, auxiliary processors are usually assigned numbers less than 1000, and users are given numbers greater than 1000. IBM has reserved the numbers 500 through 999 for customer use. You may use any number in this range without risking interference with a processor distributed by IBM in the future.

The processor number is used to make initial contact with the SVP. It is also used when one processor wants to identify another processor as a share partner. Within the APL language, processor numbers are used as the left argument of ⎕SVO **,** and may be either the right argument or the result of ⎕SVQ **.**

Global processor numbers must be unique within the processor complex. *Local* processor numbers (including those used to access TCP/IP) need only be unique within the APL session that started the processors.

Local auxiliary processors are also identified by a name that is used to start the processor. This name may appear:

- In the **ATASKS** or **RESAPS** parameters of an installation options module, if the processor is to be combined with the APL2 product
- In the **APNAMES** operand of the APL2 invocation command, or of its defaults or overrides as defined by the installation options module
- As the member name of the module in a **LOADLIB**.

For ease of association, processors included with APL2 include the processor number as a part of the processor name. You may find it helpful to adopt the same convention.

Shared Variable Identification

A shared variable has associated with it:

- A name, which identifies it between share partners.
- An index number, which identifies it in communication with the SVP.
- A sequence number, which can be used to limit matching offers.

The index number or *pershare index* (PSX) is used by the SVP to look up information about the shared variable. When a processor extends an initial offer, or matches an existing offer, the SVP assigns a specific PSX to that variable and that processor. Each partner must use its own PSX in subsequent communications with the SVP about that variable.

The *offer sequence number* (OSN) is a unique number assigned by the SVP which indicates the order in which share offers were extended. Because of limited resources, a processor may not be able to handle concurrently all offers being made to it. By limiting OSN, the processor can ask the SVP to consider only recent offers, even if earlier ones are currently unmatched. This makes it possible for the processor to acknowledge offers quickly, then switch its resources among requestors.

The variable name is used when an offer is made. For variables shared with APL sessions, the name must be alphanumeric and begin with a letter. The shared variable processor and the APL2 language processor support names up to 255 characters long. Individual auxiliary processors may impose additional restrictions. For a variable to become shared, both processors must offer it by the same name. This does not mean that the names of variables to be shared with a processor must be predefined. If your processor waits for its partner to extend an initial offer, it can issue a query to obtain the proposed name, and use that name when it extends a counter offer.

The APL user or program provides the name as the second (or only) part of the right argument to dyadic $\square SVO$, and is given the name as the result of $\square SVQ$ with a numeric right argument. If the right argument to dyadic $\square SVO$ contains two parts separated by blanks, then the first part is the name of the variable as known in the APL workspace. That name has nothing to do with the one seen by the auxiliary processor, unless the two names are identical or the second one is omitted. The two names are often referred to as the *APL name* and the *surrogate name*, respectively.

Quotas

The Shared Variable Processor has limits on:

- The number of shared variables, and
- The size of shared memory.

These limits are established separately for the global SVP (when it is started), and for the local SVP (when an APL2 session is started).

A processor may establish its own limits on number of shared variables, and SVP space for storage of their values. The processor informs the SVP of these limits as a part of the processor identification ("sign on") call. The SVP will then enforce both its own limits and any processor limits it has been given. The processor limits apply only to actions by that processor.

Note that the SVP does not guarantee either the space or the number of shares specified by the processor. Even if the values are within SVP's own limits, other processors or other variables may be using resources at the moment.

Access Control

It is often necessary for share partners to control the sequence in which they access a shared variable. If the access is not controlled, one partner can **set** a variable twice before the other can **use** the first value; or one partner can **use** a variable twice before the other can **set** a second value.

Each shared variable has associated with it a 4-bit *access control vector* (ACV) which provides a means of regulating access to the variable. Each partner presents its own version of the access control vector to the SVP. The effective, or combined, access control vector is the logical **OR** of the two. Thus each processor can impose more discipline than its partner desires, but neither can relax constraints imposed by its partner.

The meaning of each of the four bits is:

Bit 0 If 1, disallow my successive set until my partner has *accessed* the variable (either used or set it).

Bit 1 If 1, disallow my partner's successive set until I have *accessed* the variable.

Bit 2 If 1, disallow my successive use until my partner has set the variable.

Bit 3 If 1, disallow my partner's successive use until I have set the variable.

The SVP allows or disallows each access according to the variable's *access state*. The access state is defined by a 4-bit vector called the ASV, but only three states are valid:

1 0 1 0 Last access was a set by me.

0 1 0 1 Last access was a set by my partner.

0 0 1 1 Last access was a use by me or my partner. This is also the initial state before any access by either partner.

Formally, if you **AND** together the ACV and the ASV, the result will contain 1 for any operation that is currently disallowed.

Example: In reading the following, you should think of the situation as seen by one of the processors, which we will call *processor*. The other share partner we will call *partner*, realizing that each is really both a processor and a partner. The note below will explain why it is important to restrict your viewpoint to one side of the partnership.

1. *Processor* wants its own use and set controlled.

   Access Control Vector contribution: `1 0 1 0`

2. *Partner* wants its use and its partner's use controlled.

   Access Control Vector contribution: `0 0 1 1`

   Combined Access Control Vector: `1 0 1 1`

3. *Partner* sets the variable

   Access State Vector: `0 1 0 1`

   (This is still as seen by *processor*.)

4. The **AND** of the two vectors is now `0 0 0 1`, which provides the following information:

   Bit 0 Off, so *processor* is allowed to set the variable. One partner can always choose to ignore a value set by the other, and replace it with a new value. There is, separate from access control, a switch used to indicate whether the processor wishes to do this.

   Bit 1 Off, so *partner* is allowed to set the variable again. Note that neither program asked that *partner* sets be controlled.

   Bit 2 Off, so *processor* may currently use the variable.

   Bit 3 On, so *partner* may not use the variable until *processor* has set it.

Note: The SVP transposes bits in both the access control vector and the access state vector when presenting them to each partner. This is done to make the definitions consistent as seen from both sides. You may want to think of the 4-bit objects as 2 by 2 matrices, like this:

My set	Partner set
My use	Partner use

Now imagine the matrix as Boolean numbers printed on a piece of glass. If the glass is held up between the two share partners, each will see it with the proper transpositions. (The creators of the Arabic numerals showed remarkable foresight in designing 0 and 1 so that they would be readable from both sides of the glass.)

Event Signalling and Event Control Blocks

The SVP uses *event control blocks* (ECBs) to *signal* between share partners. For example, one processor may try to set a variable, but be denied because of the current access state and access control vectors. The SVP will attempt to signal that processor when its share partner uses the variable, since that action makes it possible for the first processor to set the variable.

SVP signals are sent by *posting* an ECB. This is an operating system (rather than an APL) concept, and is accomplished using system services appropriate to the environment. The auxiliary processor may *wait* for an ECB (or one of a set of ECBs) to be posted. Global auxiliary processors must use operating system **WAIT** services for this. Local auxiliary processors will normally use the service described in “PW: Wait for an Event” on page 68, except that local processors under TSO may choose the MVS **WAIT** service if they prefer.

A Processor must always provide an ECB when it first identifies itself to the SVP. The address of this ECB is retained by the SVP, which may post the ECB at any time. A processor will normally also provide a unique ECB for each shared variable which it offers or matches. The processor may choose not to provide a separate ECB for each share offer. In this case all subsequent signals which would normally post the ECB for that variable will instead post the processor ECB.

The SVP may signal a processor for a number of reasons. Most of these reasons can be associated with an existing share offer previously made by the processor, and the signalling is accomplished by posting the share ECB (if available). Signals which cannot be associated with an available share ECB are sent by posting the processor ECB.

ECB Post Codes: The SVP uses a set of flags to indicate the type of a signal that is being sent. In some cases, two or more separate signals may occur that would be posted to the same ECB without intervening action required by the partner being posted. Depending on the system, the signal flags may be **OR**ed, or only one of the flags may be set and other signals will appear to be lost. The processor should never depend on a particular flag being set, but should instead attempt the next desired operation or ask for state information as described in “CSVSTATE: State” on page 90.

The signal flags are placed in the last byte of the ECB and have the following meanings:

`X'01'` Coupling Change: Match or Retract has occurred for a previously offered variable, or an Offer has been made to the processor. The match or Retract signals will be posted in the share ECB if one is available.

`X'02'` Set Access Control Vector performed. This may or may not have affected the combined Access Control Matrix.

`X'04'` Variable Set or Use Action. These are discussed further in “Set or Use Signals” on page 10. Not all of the cases result in signals.

X'10' Resource Available. A previous request was rejected for some reason such as:

- Too many concurrent processors or variables
- A storage shortage in the SVP
- The partner held control of the variable.

The resource may now be available, so the processor should re-try the request if desired.

X'20' The processor is being asked to sign off. This signal is sent when the SVP is terminating. For local processors, it means that the user is exiting from APL2. For global processors, it means the system operator is shutting the global shared variable processor down.

Set or Use Signals: These are best considered from the viewpoint of the processor taking the action, rather than the processor being signalled. The decision as to whether to signal a partner is directly inferable by the SVP from the:

- Merged Access Control Vector (ACV), and
- Access State Vector (ASV) at the start of the request, and
- Processor request.

The ACV and ASV have been described in "Access Control" on page 7. The only processor requests discussed in this section are **set** and **use**, though other requests can cause signalling, as "ECB Post Codes" on page 9 indicates. For conformability with the ACV and ASV, we define a *request vector* (RQV) having the following possible values:

0 1 1 0 variable use

1 0 0 1 variable set

A signal will be generated if the following APL expression evaluates as 1:

$$1 \in RQV \wedge ACV \wedge ASV$$

In words, this means that if you line the three vectors up and scan down the four columns of 1's and 0's, then a signal will be generated if any column has all 1's. (It doesn't matter which column has the 1's.)

In the same way that the combined access control can be viewed as a 2 by 2 matrix specifying what is controlled:

My set	Partner set
My reference	Partner reference

it can also be viewed as a matrix specifying that my partner is to be signalled on my:

Failed set	Successful reference
Failed reference	Successful set

Note that the failed actions on the left need to cause signals so that my partner can take action to relieve the corresponding control. The successful actions on the right cause signals because they may have relieved a corresponding control encountered by my partner.

Chapter 2. Associated Processor External Routine Concepts

Associated processors are synchronous subroutines of the APL2 interpreter. They provide support for *external objects*; that is arrays, functions, or operators which are known within a workspace but are not actually part of the workspace.

External objects are made known to a workspace by using a *name association* system function, ⎕*NA*. ⎕*NA* is described in *APL2 Programming: Language Reference*. It provides an association between a name in a workspace and a (perhaps different) name as recognized by a particular associated processor.

Note that name association does not involve the use of shared variables. Although some of the concepts are similar, the differences are striking:

- Name association routines are synchronous subroutines; shared variables are processed by independent, asynchronous programs.
- Name association provides variable, function, and operator support; shared variables are just that: variables.
- Once a name association has been made in a workspace, it persists across)*SAVE* or)*OUT* of that workspace, and subsequent)*LOAD*,)*COPY*, and)*IN* of the whole workspace or the names that were associated. Shared variables apply only to the active workspace, and must be reestablished by user program action each time the workspace is loaded.

As just indicated, name association persists as viewed by the APL programmer or user. What really happens, though, is that it is automatically reestablished whenever required. APL2 retains the ⎕*NA* arguments as descriptive information associated with the object. These arguments include:

- An associated processor number,
- A name to pass to the associated processor, and
- Additional descriptive material that the processor may need to resolve the association.

The associated processor may then resolve the name as a program written in APL2, FORTRAN, REXX, C/370, PL/I, assembler language, or perhaps other languages.

In many cases the programs provide utility services, and need not be aware that they were called from APL2. The details of calling this class of programs is provided in *APL2 Programming: System Services Reference*.

In other cases the programs may be written specifically as extensions of the APL2 language or as high performance replacements for APL2 algorithms. This class of programs works much more closely with arrays, functions, and operators that exist in the active workspace. These are the programs that will use the interfaces defined here.

All of these (APL-specific) programs are defined to processor 11 as functions or operators by means of :LINK tag in a routine description. See *APL2 Programming: System Services Reference* for these descriptions. The programs are called *associated processor external routines*, or simply *external routines*. We use the shorter term in the remainder of this manual.

External routines can mimic APL functions and operators. They can have any valence a normal APL2 routine can have and normally produce an explicit result.

`:LINK.FUNCTION` routines are designed to be written in assembler language and are given access to processor 11 and APL2 services, the argument data descriptors, and the data itself. All of their arguments and results are in common data representation (CDR) format or are represented by tokens. A service is provided to convert data represented by a token to a CDR. Arguments are passed as pointer format CDRs. Results can be returned in any of the supported CDR formats. See Chapter 3, “APL2 Data Representation” on page 13 for a complete description of CDRs.

`:LINK.OBJECT` and `:LINK.FORTRAN` routines can be written in FORTRAN, assembler, C/370, PL/I, or other languages. They are given access only to processor 11 services. Arguments are passed to them using standard OS or FORTRAN conventions. Results are returned using the same conventions.

Chapter 3. APL2 Data Representation

Common data representation (CDR) is a convention for the construction and interpretation of self-describing data objects that can be communicated between APL processors.

In the CDR, a data object is assumed to be an APL array and is classified as either a simple or a general object:

- A simple object is an array of simple homogeneous scalar values.
- A general object is an object that is not a simple object. These are arrays containing mixed character and numeric data or an array containing arrays as items.

Common data representation recognizes three forms, but one of them is retained only for compatibility with earlier releases. The two forms you will be dealing with are:

- The *dense* form in which the CDR object contains all of the data in addition to all descriptive information about it.
- The *pointer* form in which the CDR object contains all of the descriptive information about the data but has pointers to part or all of the data itself.

CDR objects passed from the SVP to an auxiliary processor are passed in dense form. CDR objects passed from processor 11 to external routines using FUNCTION linkage are passed in pointer form. Auxiliary processors and external routines may pass CDR objects to the Shared Variable Processor or to processor 11 in dense or pointer form.

A CDR object is divided into four sections:

- A header section containing information about the CDR itself.
- A descriptor section, which describes the size and shape of the object and how it is represented.
- An optional pointer section, which contains the information about the actual location and length of the data.
- An optional data section, which contains all (dense form) or part (pointer form) of the data.

The CDR for a simple object contains the header, one descriptor, and either a pointer or the data, in that order. The data is stored in ravelled order. (This is row by row, not column by column, in case you are used to FORTRAN conventions.)

CDRs for general objects contain the same sections in the same order (though they may contain both pointer and data sections), but the sections are somewhat more complex.

- There are always multiple descriptors.
- There are frequently a number of pointers.
- Data aggregates are usually laid out one after the other with no regard for data types or normal boundary alignment.

The descriptor section always begins with information about the general object itself. This is followed by descriptors for each item of the general object. For a general object that contains other general objects, a recursive descriptor structure is used. Figure 2 is an example of such an object.

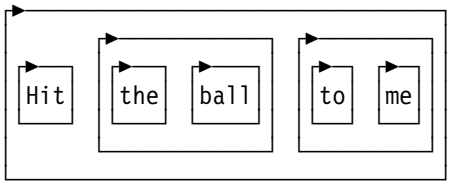


Figure 2. Example of a General Object

The arrows in this example are used by the *DISPLAY* function to provide shape information. A full description of this representation is available in *Using the Supplied Routines* and in the *DISPLAY* workspace. The object shown here is a three-element vector, containing:

1. A simple character vector “Hit.”
2. A two-element vector, which in turn contains:
   a. A simple character vector “the.”
   b. A simple character vector “ball.”
3. Another two-element vector, which in turn contains:
   a. A simple character vector “to.”
   b. A simple character vector “me.”

The CDR descriptors appear in what would formally be called a *ravelled, left-list order*. This is really the order used in the numbered and lettered lists just given. The *PFA* external function, described in *Using the Supplied Routines*, builds a pattern comparable to CDR descriptors. You may find it quite useful in understanding CDR structures.

For this example there would be eight descriptors, one for each box in the picture. They would appear in the following order:

(general) hit (general) the ball (general) to me

where the term “(general)” is used to indicate a nonsimple array. Notice that each (general) descriptor is followed by descriptors of the items within that array. If one of those items is in turn a (general), then the current list is interrupted to describe its items.

Even an empty general object will contain a descriptor for the general object followed by at least one more descriptor. The additional descriptor or descriptors define a prototype structure for the general object.

The pointer and data sections follow the order established in the descriptor section, but never contain any entries associated with general objects, empty items, or prototypes.

Representation Types and Lengths

There are several ways in which data can be represented in the CDR. Each representation has associated with it a *representation type* (RT), a *representation length* (RL), and rank and shape information.

The representations currently defined are intended to support the requirements of APL but would be useful in many other environments. Figure 3 shows the combinations of RT and RL that are defined.

Figure 3 (Page 1 of 2). CDR RT/RL Combinations

RT	RL	Scale	Description
'B'	1	bit	Boolean integers (0 and 1).
'B'	4	bits	Hexadecimal integers (0 thru 15).
'B'	8	bits	Byte integers (0 thru 255).
'I'	2	bytes	Signed fixed-point integers. S/370 or S/390* halfword integers (¯32768 through 32767).
'I'	4	bytes	Signed fixed-point integers. S/370 or S/390 fullword integers (¯2147483648 through 2147483647).
'E'	4	bytes	Real floating-point numbers. S/370 or S/390 single precision floating numbers (6 hex digit precision).
'E'	8	bytes	Real floating-point numbers. S/370 or S/390 double precision floating numbers (14 hex digit precision).
'E'	16	bytes	Real floating-point numbers. S/370 or S/390 extended precision floating numbers (30 hex digit precision).
'J'	8	bytes	Complex floating-point numbers. Two single precision floating values representing real and imaginary parts of the complex value.
'J'	16	bytes	Complex floating-point numbers. Two double precision floating values representing real and imaginary parts of the complex value.
'J'	32	bytes	Complex floating-point numbers. Two extended precision floating values representing real and imaginary parts of the complex value.
'C'	1	byte	Character. S/370 or S/390 EBCDIC encoding with APL graphics assumed.
'C'	4	bytes	Character extended, for double-byte character sets. The first two bytes are a character set identifier; the last two bytes are a code point within that character set. APL/EBCDIC characters have X'00' in the first 3 bytes.
'A'	8	bytes	Integer progression. Two fullword integers representing the first value and the increment between values in a string of values. (The number of values in the string is stored in the descriptor for the object.)
'P'	*n*	bytes	Packed decimal format, where $1 \le n \le 16$. *n* is the width in bytes of the packed decimal field.
'Z'	*n*	bytes	Zoned decimal format, where $1 \le n \le 16$. *n* is the width in bytes of the zoned decimal field.

Figure 3 (Page 2 of 2). CDR RT/RL Combinations			
RT	**RL**	**Scale**	**Description**
'X'	0	byte	Filler. This type is provided for use within a general object. It indicates that bytes in the data section are simply fillers and are to be otherwise ignored. The number of bytes to be ignored is the number of items defined by the descriptor.
'G'	0		General object. The data associated with this is described in subsequent descriptors.

CDR Format

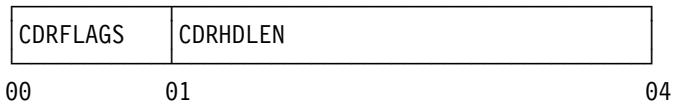
The format of an object in the CDR is divided into four parts:

1. The header section
2. The descriptor section
3. The pointer section
4. The data section

The parts of a CDR object will be contiguous and in the above order, although not all types will have all sections. Dense form CDRs will have a header, one or more descriptors, and a data section. Pointer form CDRs will have a header, one or more descriptors, one or more pointers, and optionally a data section.

The CDR must start on a word boundary. Since all parts except the data section are multiples of four bytes in length, all sections will also start on fullword boundaries.

Header Section



CDRFLAGS The bits in the flag byte have the following definitions:

X'80' **CDRID**: Always on, used to distinguish a CDR object from earlier forms of shared variables used in APLSV and VS APL.

X'20' **CDRPTRF**: On if the data is in pointer form. See “Pointer Section” on page 17 for more information.

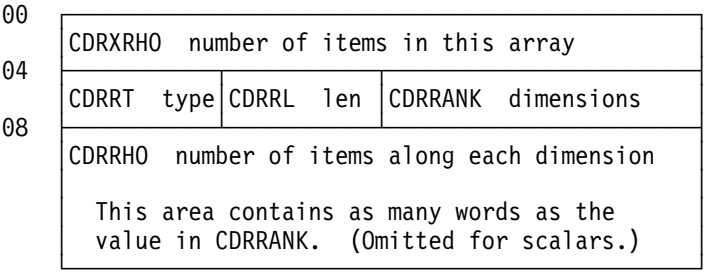
All other bits in the flag byte are reserved for future use, and must be set to zero.

CDRHDLEN The length of the header and descriptor sections together. Thus this may be used as an offset to the pointer section if **CDRPTRF** is on, or to the data section if it is off.

Descriptor Section

The descriptor section of a CDR contains a descriptor for the object as a whole and, for mixed or nested objects, a descriptor section for each item of the object. Thus a simple array in the CDR format will contain a single descriptor, while a general array will contain a recursive structure of descriptors. In a general array with nested subarrays, the descriptors (and the data) will be stored in *ravelled left-list order*.

Each descriptor in the CDR has the following format:



CDRXRHO is the total number of items defined by the descriptor. In APL terminology, **XRHO** means ×/*RHO*.

CDRRT and **CDRRL** are the representation type and the representation length, respectively. **CDRRT** is a character code. **CDRRL** is the binary length of one ***element***. For example, `C'I',FL1'4'` represents a fullword integer. (**I**4 is often used as a shorthand notation for this, but you are cautioned to remember that **I** is character, while 4 is not.) See "Representation Types and Lengths" on page 15 for a complete definition of **RT/RL** combinations.

The dimension vector, **CDRRHO**, uses one word for each dimension, where **CDRRANK** is the number of dimensions. For scalars, **CDRRANK** is 0. Thus there is no dimension vector and the descriptor consists of 2 words only.

For nested arrays (arrays whose items are themselves arrays), **CDRRT** is always **G** and **CDRRL** is 0. Each item of such an array will have its own descriptor following the **G**-type descriptor. Since those descriptors may in turn be **G**-type (with their own subordinate descriptors), the item descriptors of a **G**-type array may not be contiguous.

Unnested, but mixed (nonhomogeneous), arrays also use a **G**-type descriptor with subordinate descriptors for each item. In this case the subordinates will all describe scalars.

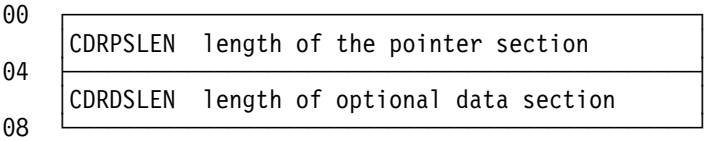
Finally, empty nested or mixed arrays also have a subordinate descriptor which defines the prototype of the array.

Note: The descriptor for the n^{th} item in a nonsimple array may be located by beginning beyond the array descriptor with a count of n-1, and stepping across descriptors. For each descriptor,

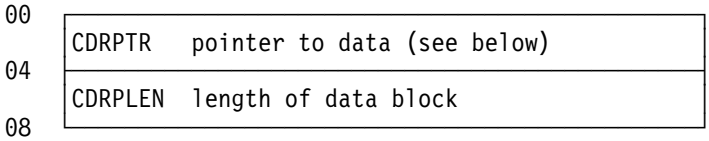
- If **RT=X**, just skip it.
- Else if **RT=G** then add **CDRXRHO** to the count (but add 1 if **CDRXRHO=0**).
- Else subtract one from the count.

Pointer Section

The pointer section exists if, and only if, **CDRPTRF** is set in the CDR header. This section contains one pointer description segment followed by one or more data pointer segments. The pointer description segment has this format:



Each data pointer segment takes the following form:



Pointers may be positive, negative, or zero.

- Positive pointers are data tokens, not addresses. They may be passed from an owning routine to some other routine, and later returned to the owner, but only the owner knows how to interpret them to actually locate the data. Data tokens are used by the APL2 interpreter in passing arguments to external routines.
- Negative pointers are 31-bit absolute addresses of data, but with the high bit set. (They are not complemented addresses.) Note that the APL2 interpreter often moves data from one place to another within the workspace to coalesce free storage areas. Because of this, absolute addresses of data within the workspace are only valid for very limited periods of time.
- A zero pointer indicates that the corresponding piece of data is found in the data section of the CDR. Only data corresponding to zero pointers is stored as a part of the CDR. It is stored in byte-aligned areas in the order defined by those data pointer segments.

Each block of data described by a data pointer segment must contain one or more complete simple arrays. CDRs passed to External Routines use a separate pointer for each simple array.

Data Section

This area contains all data defined by dense form CDRs (**CDRPTRF** off) or any zero-pointer data defined by pointer form CDRs. In either case the data is densely packed, with each simple array beginning on a byte boundary.

In particular, dense packing for **B**-type arrays means that multiple elements are stored per byte unless **RL=8**.

Note that **X**-type descriptors can be used to skip bytes in this area. This makes it possible to maintain halfword, fullword, or doubleword alignment where this would be useful.

AP2CDR Mapping Macro

Invoking the Macro

```
AP2CDR [TYPE=CSECT][,PRE=prefix][,DOC=NO][,EQU=NO]
```

TYPE=CSECT:

An optional parameter which indicates that the parameter block is to be mapped inline without generating a DSECT. The default is **TYPE=DSECT**.

PRE=*prefix*:

A three character prefix used for all generated labels and the DSECT name. The default is **PRE=CDR**.

DOC=NO:

An optional parameter which reduces the size of the listing by not generating full explanations of the CDR format and usage. The default is **DOC=YES**.

EQU=NO:

An optional parameter which omits definitions for the CDRRT and CDRRL codes. These definitions are generated using names that begin with RT and RL, and are not controlled by the PRE= parameter. If you expand AP2CDR more than once in one assembly, you must specify **EQU=NO** on all but one of the expansions. The default is **EQU=YES**.

Typical Expansion

```
         AP2CDR
CDR      DSECT
*
*        CDR HEADER SECTION
*
CDRDLEN  DS    0F                  LENGTH OF HEADER + DESCRIPTOR
*        NOTE: HIGH BYTE MUST BE CLEARED IF ABOVE FIELD IS USED.
CDRFLAGS DS    X                   FLAG BITS
CDRID    EQU   X'80'               CDR IDENTIFICATION FLAG
CDRSCAT  EQU   X'40'               SCATTERED FORMAT
CDRPTRF  EQU   X'20'               POINTER FORMAT
*        NOTE: SCAT IS RETAINED FOR RELEASE 1 COMPATIBILITY
*
CDRHDLEN DS    FL3                 LENGTH OF HEADER + DESCRIPTOR
*        CDR DESCRIPTOR SECTION
*
CDRDES   DS    0F                  ITEM DESCRIPTOR
CDRXRHO  DS    F                   NUMBER OF ELEMENTS
*
CDRRTRL  DS    0XL2                TYPE AND LENGTH
CDRRT    DS    C                   REPRESENTATION TYPE
*        NOTE: SEE RT.. CODES DEFINED BELOW
CDRRL    DS    X                   REPRESENTATION LENGTH
*        NOTE: SEE RL.. CODES DEFINED BELOW
CDRRANK  DS    H                   RANK
CDRRHO   DS    0F                  DIMENSION VECTOR
*        POINTER SECTION
*
CDRDATA  DSECT
CDRPSLEN DS    F                   LENGTH OF POINTER SECTION
CDRDSLEN DS    F                   LENGTH OF DATA SECTION
CDRPTR   DS    A                   DATA POINTER
CDRPTRB  EQU   X'80'               SET IF ABSOLUTE ADDRESS
CDRPLEN  DS    F                   DATA LENGTH
```

```
*        CDR REPRESENTATION TYPES
*
RTMIN    EQU   C'A'               LOWEST HEX VALUE FOR RT TYPE EQUS
RTB      EQU   C'B'               UNSIGNED INTEGER (BOOLEAN)
RTI      EQU   C'I'               SIGNED INTEGER
RTE      EQU   C'E'               REAL NUMBER
RTJ      EQU   C'J'               COMPLEX NUMBER
RTC      EQU   C'C'               CHARACTER DATA
RTA      EQU   C'A'               ARITHMETIC PROGRESSION
RTP      EQU   C'P'               PACKED DECIMAL
RTZ      EQU   C'Z'               ZONED DECIMAL
RTX      EQU   C'X'               FILLER
RTG      EQU   C'G'               GENERAL OBJECT
RTF      EQU   C'F'               FUNCTION/OPERATOR
RTMAX    EQU   C'Z'               HIGHEXT HEX VALUE FOR RT TYPE EQUS
*        CDR REPRESENTATION LENGTHS
*
RL1      EQU   1                  1 BIT (BOOLEAN) ELSE 1 BYTE
RL2      EQU   2                  2 BYTES
RL4      EQU   8                  4 BITS (BOOLEAN) ELSE 4 BYTES
RL8      EQU   8                  8 BITS (BOOLEAN) ELSE 8 BYTES
RL16     EQU   16                 16 BYTES
RL32     EQU   32                 32 BYTES
RLN      EQU   0                  UNDEFINED OR VARIABLE LENGTH
*        VALID COMBINATIONS OF REPRESENTATION TYPE AND LENGTH
*
RTLB1    EQU   RL1+RTB*256        1 BIT BOOLEAN
RTLB4    EQU   RL4+RTB*256        4 BIT BOOLEAN
RTLB8    EQU   RL8+RTB*256        8 BIT BOOLEAN
RTLI2    EQU   RL2+RTI*256        INTEGER HALFWORD
RTLI4    EQU   RL4+RTI*256        INTEGER FULLWORD
RTLE4    EQU   RL4+RTE*256        SINGLE PRECISION FLOATING POINT
RTLE8    EQU   RL8+RTE*256        DOUBLE PRECISION FLOATING POINT
RTLE16   EQU   RL16+RTE*256       EXTENDED PRECISION FLOATING POINT
RTLJ8    EQU   RL8+RTJ*256        DOUBLE PRECISION COMPLEX
RTLJ16   EQU   RL16+RTJ*256       DOUBLE PRECISION COMPLEX
RTLJ32   EQU   RL32+RTJ*256       DOUBLE PRECISION COMPLEX
RTLA8    EQU   RL8+RTA*256        FULLWORD INTEGER PROGRESSION
RTLC1    EQU   RL1+RTC*256        CHARACTER BYTE
RTLC4    EQU   RL4+RTC*256        CHARACTER FULLWORD
RTLPN    EQU   RLN+RTP*256        VARIABLE LENGTH PACKED DECIMAL
RTLZN    EQU   RLN+RTZ*256        VARIABLE LENGTH ZONED DECIMAL
RTLXN    EQU   RLN+RTX*256        VARIABLE LENGTH FILLER
RTLGN    EQU   RLN+RTG*256        GENERAL OBJECT
RTLFN    EQU   RLN+RTF*256        FUNCTION/OPERATOR
RTLF1    EQU   RL1+RTF*256        EXTERNAL NAME
```

Note: If you need to use the AP2CDR mapping macro, see Appendix C, “Macros Intended for Customer Use” on page 182 for the information you need to get from your system programmer.

Part Two: Interfaces and Services from APL2

Chapter 4. Entry and Exit Conditions

Auxiliary processors and processor 11 external routines have somewhat different entry and exit conditions. This is necessary since Auxiliary Processors are separate asynchronous tasks, but external routines are subroutines called (indirectly) by the APL2 interpreter.

Local Auxiliary Processor Entry and Exit

Local auxiliary processors are started by the APL2 executor, normally during user invocation of APL2. When an auxiliary processor is started, it is given control using a standard **CALL** linkage, as follows:

- Register 13 points to a save area.
- Register 14 contains a return address.
- Register 15 contains the processor entry point.
- Register 1 points to a parameter list, described below.

In an MVS/XA* environment, the RMODE and initial AMODE of the auxiliary processor (AP) are controlled by the way it was link-edited. For APs linked with the executor, AMODE=24 and RMODE=24.

In a VM environment supporting 31-bit addressing, the RMODE and initial AMODE of an auxiliary processor are controlled by the way it is defined in the APL2 options module and/or is link-edited:

- AP entry points defined in the RESAPS option are entered AMODE=24. The APs must be link-edited in an APL2 LOADLIB which has RMODE=24.
- AP entry points defined in the ATASKS option are entered AMODE=31. The APs must be link-edited in an APL2 LOADLIB, but a LOADLIB with any AMODE or RMODE may be used.
- AP entry points defined in the RESEPS option are declared as nucleus extensions with AMODE=31. The entry point names must also be included in the APNAMES invocation option, and are entered AMODE=31. The APs must be link-edited in an APL2 LOADLIB, but a LOADLIB with any AMODE or RMODE may be used.
- AP entry points named in the APNAMES invocation option which have been declared as nucleus extensions are started in the AMODE declared for the nucleus extension.
- AP entry points named in the APNAMES invocation option which are loaded from the LOADLIB FILEDEF'ed to the name AP2LOAD are loaded and started based on the AMODE and RMODE assigned when they were link-edited.

Any AP may change its own AMODE as desired. Except where stated otherwise, services can be requested in either AMODE with any data appropriate to the mode used.

The parameter list is a series of fullword addresses with the high-order bit set on the last word in the list. See Figure 4 on page 23.

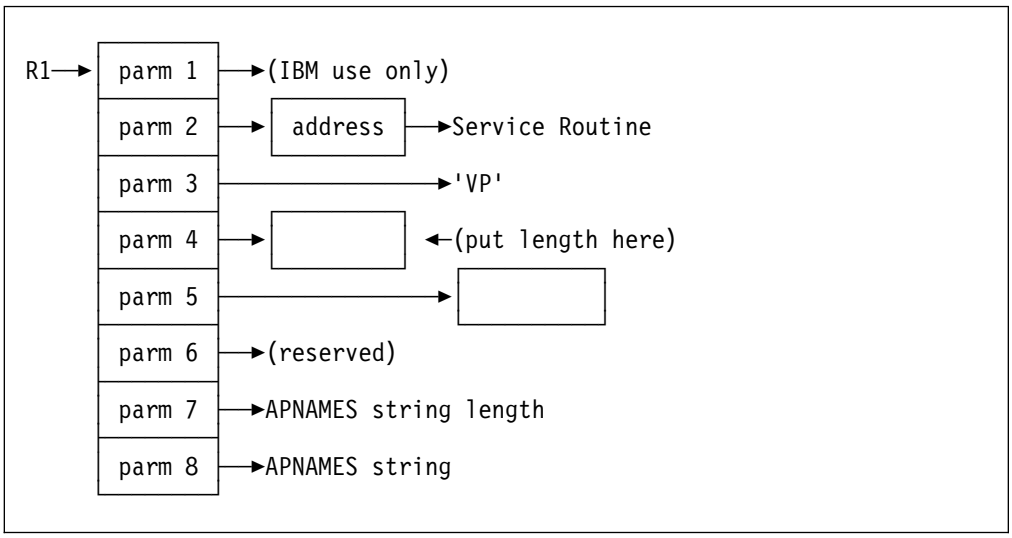


Figure 4. Local Auxiliary Processor Input Parameters

`parm 1` Used only by VS APL auxiliary processor compatibility support.

`parm 2` A pointer to a service routine which may be called by the auxiliary processor. This service routine supports the services described in subsequent chapters of this part.

`parm 3` This is the beginning of a model parameter list for the virtual storage service (see "VP: Get Process Storage" on page 97). This helps solve the bootstrapping problem of needing storage for the request that obtains storage.

`parm 4` Fill in here the number of bytes of dynamic storage you need. Then point register 1 at the third word in the parameter list (the one that points at **VP**) and call the service routine.

`parm 5` After calling the service routine with the **VP** request as described immediately above, this parameter will be returned as a pointer to the requested storage area.

`parm 6` Reserved.

`parm 7` When the **APNAMES** invocation option is used to specify auxiliary processors, an optional processor parameter string may be supplied:

APNAMES*(name(string))*

The length of that string (if any) will be provided here. A length of zero indicates that no string was provided.

`parm 8` The parameter string provided in the **APNAMES** invocation option. This parameter is present only if the previous one is non-zero.

If possible, your processor should be written so that it does not depend on the contents of any other registers at entry. Many processors provided by IBM do assume that register 10 points to an APL control block called the PTH. But be warned that neither this register nor the format of the control block are guaranteed; they may be changed for any future releases or versions of the APL2 product.

Auxiliary Processor Exit

Auxiliary processors should terminate when they receive an CSVENA return code from an SVP service, or observe the "sign off" signal described in "ECB Post Codes" on page 9. This signal will always be posted in the processor ECB. Processors should break their connection with the SVP before terminating. See "CSVOFF: Signoff" on page 73. On normal termination, registers 2 thru 14 must be as at entry, and the processor must return to the address in register 14.

An abnormal termination will occur if an unrecovered program check or ABEND occurs in the auxiliary processor task. Processors may recover from all program checks and most abends by using the services described in "EX: Set or Clear an ABEND Exit" on page 46.

Global Auxiliary Processor Entry and Exit

Global auxiliary processors are started as separate jobs or virtual machines, normally during operating system initialization or by operator command. They are given control directly by the operating system, and must in turn contact the APL2 shared variable processor. They do this by calling the appropriate APL2 interface initialization routine:

`AP2TAPV2` for APL2 interface under TSO
`AP2VAPV2` for APL2 interface under CMS

The interface initialization routine is called using a standard **BALR 14,15** linkage. When that routine is called, register 13 must point to an 18-word OS save area, and register 1 must point to a parameter list as shown in Figure 5.

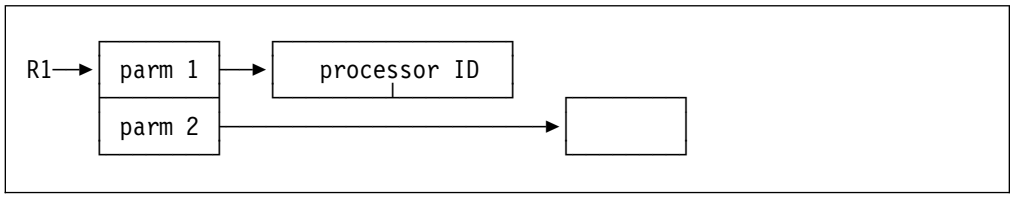


Figure 5. Global Auxiliary Processor Initialization Interface

`parm 1` An 8-byte area which must contain the proposed Processor ID to be used by the auxiliary processor. This field may be modified on return. If so, the modified value must be used for all subsequent requests. Note that an APL2 session can share variables with the processor only if the last four bytes of the ID are binary zeros. APL2 treats the first four bytes as a fullword binary number.

`parm 2` A fullword in which a pointer to the service routine will be returned. This service routine supports the services described in subsequent chapters of this part.

On return from the interface initialization routine, register 15 is zero if access to the SVP is permitted or one if it is denied. When the return code is zero, the parameters will have been modified as indicated above.

:LINK.OBJECT Routine Entry

Whenever a :LINK.OBJECT routine is called, it is given control using a standard CALL linkage, as follows:

- Register 13 points to a save area
- Register 14 contains its return address
- Register 15 contains its entry point
- Register 1 points to a parameter list, described below.

The save area pointed to by register 13 is 36 fullwords long. This length is provided for interface management routines that need to call external routines and regain control before returning to processor 11.

Note: No ECV is provided to :LINK.OBJECT routines.

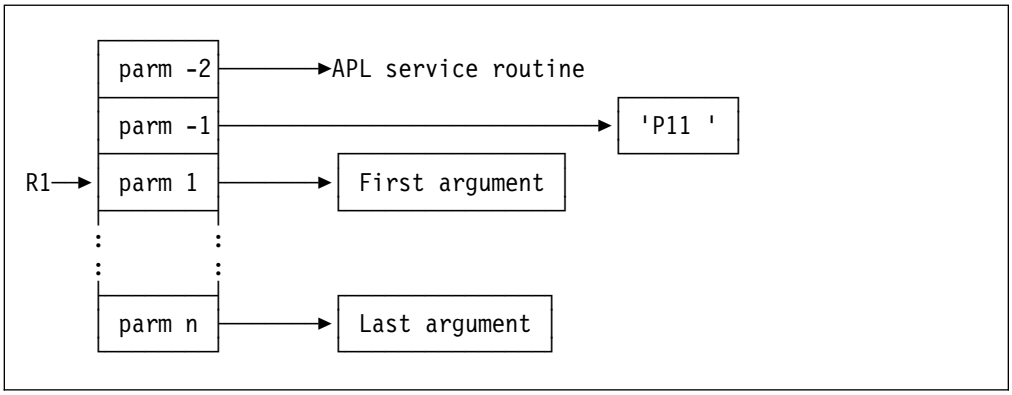


Figure 6. :LINK.OBJECT Routine Invocation Parameter List

:LINK.FORTRAN Routine Entry

Whenever a :LINK.FORTRAN routine is called, it is given control using a standard CALL linkage, as follows:

- Register 13 points to a save area
- Register 14 contains its return address
- Register 15 contains its entry point
- Register 1 points to a parameter list, described below.

The save area pointed to by register 13 is 36 fullwords long. This length is provided for interface management routines that need to call external routines and regain control before returning to processor 11.

Note: No ECV is provided to :LINK.FORTRAN routines.

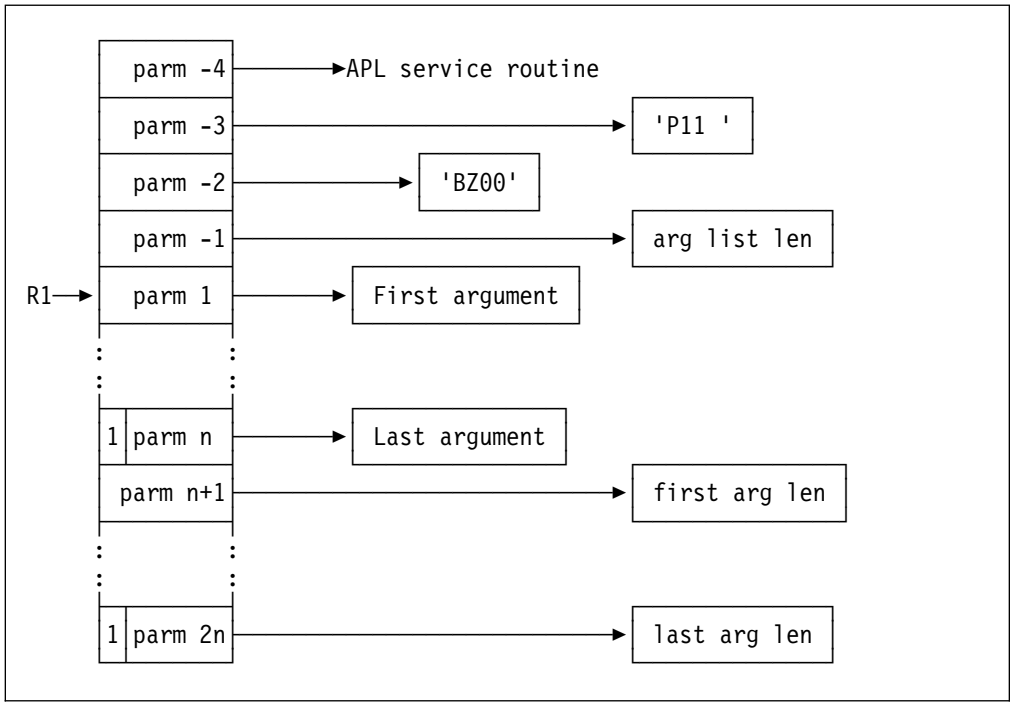


Figure 7. :LINK.FORTRAN Routine Invocation Parameter List

The :LINK.FORTRAN routines' parameter lists include length information that is not found in :LINK.OBJECT routine parameter lists. This information is provided because FORTRAN expects this information to be available for subroutine arguments.

:LINK.FUNCTION Routine Entry and Exit

The primary component of the interface to :LINK.FUNCTION routines is a control block called the *external control vector* (ECV). The first subsection below describes that control block. The next subsection discusses the primary call to an external routine, the external call interface used to respond to a function call. Then the conventions for returning from an external call are described. Finally, the delete linkage call is described.

The External Control Vector (ECV)

Each time a :LINK.FUNCTION routine is entered, it is given the address of an ECV which defines the arguments to the routine and provides a way to obtain additional information and services from the interpreter. At the completion of the routine execution the ECV is also used to return a result to the interpreter. The ECV is composed of a basic section (Figure 8), which is always used, and an extension (Figure 9 on page 28), which is used for the external call interface. When making use of APL service routines, you must use the ECV provided by processor 11. Processor 11 rejects any ECV that it did not provide.

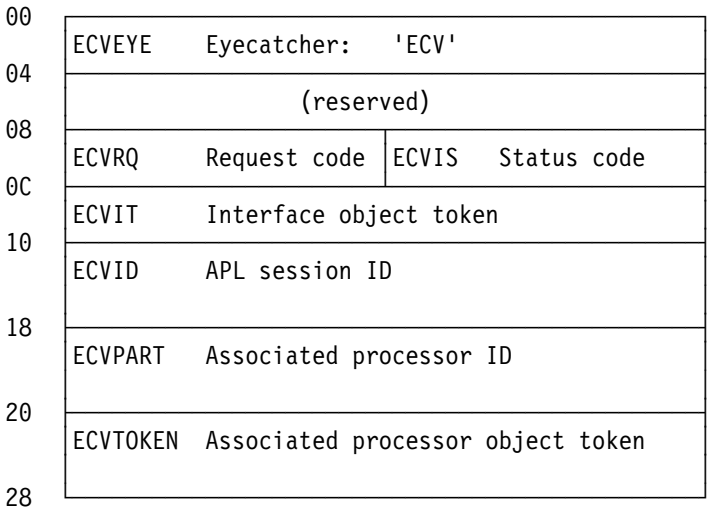


Figure 8. ECV Basic Section

28	ECVXCET	Result event type (2 halfwords)
2C	ECVXCDRZ	Result CDR object token
30	ECVXCDRL	Left argument CDR address
34	ECVXCDRR	Right argument CDR address
38	ECVXPRQT	Service request object token
3C	ECVXPRQP	Service request parameter
40	ECVXPRQX	Service request extra parameter
44	ECVXTLA	Left argument object token
48	ECVXTLF	Left function object token
4C	ECVXTOP	Operator object token
50	ECVXTRF	Right function object token
54	ECVXTRA	Right argument object token
58	ECVXRLOC	Relocation count
5C	ECVXERMS	Quad–ES message object token
60		

Figure 9. ECV External Call Extension

Note: On entry to the :LINK.FUNCTION routine, fields in the ECV noted as "object token" will contain values used internally by APL2. These object tokens are always positive, and may be returned unchanged or may be redeemed for absolute addresses by issuing an XB service request to the APL2 service routine. On exit from the :LINK.FUNCTION routine, these same ECV fields may contain absolute addresses with the high-order bit (X'80') on or APL object tokens with the high-order bit off. Object tokens are either

- CDR absolute addresses with the high order bit on,
- Tokens received from the XG service identifying storage containing a CDR or
- Tokens for APL2 objects received during the function call or through the XB, XE, XF or XG services.

Any form of object token may be passed on service calls or returned to APL2 in the ECV.

Individual fields in this block are described as needed in the following subsections. Other fields in the ECV are used in conjunction with the services defined in Chapter 14, "Code X__: External Call Services" on page 101. Additional details and a mapping for the ECV may be found in the AP2ECV macro, which is distributed with APL2.

Note: If you need to use the AP2ECV mapping macro, see Appendix C, "Macros Intended for Customer Use" on page 182 for the information you need to get from your system programmer.

:LINK.FUNCTION Routine Entry: External Call

Whenever a `:LINK.FUNCTION` routine is called, it is given control using a standard **CALL** linkage, as follows:

- Register 13 points to a save area.
- Register 14 contains its return address.
- Register 15 contains its entry point.
- Register 1 points to a parameter list, described below.

The save area pointed to by register 13 is 36 fullwords long. This length is provided for interface management routines that need to call external routines and regain control before returning to processor 11.

In an XA or ESA* environment, the **RMODE** and initial **AMODE** of the module are controlled by the way it was link-edited. The module may change its own **AMODE** as desired. Except where stated otherwise, the services can be requested in either **AMODE**, with any data appropriate to the mode used. Normally, however, **AMODE=31** will be required to access either workspace data or argument data passed in pointer form CDRs.

Figure 10 shows the format of the parameter list.

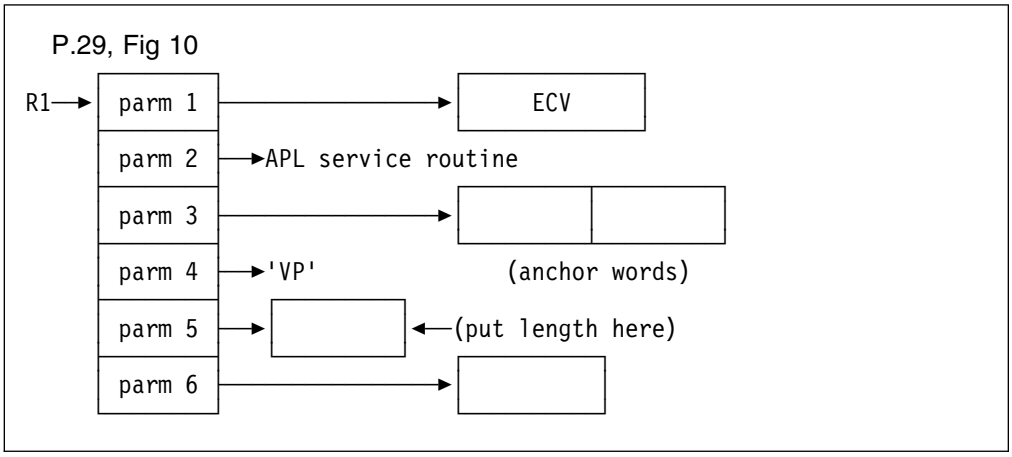


Figure 10. :LINK.FUNCTION Routine Invocation Parameter List

The data pointed to by the addresses in the parameter list is defined as follows:

1. The ECV defined in “The External Control Vector (ECV)” on page 27. ECV fields used on the call to the external routine are described on page 30.
2. A fullword containing the address of the APL service routine. The service routine can be called to use all the services defined in the following chapters.
3. A doubleword that contains binary zeros on the initial call to the external routine. Data placed in the doubleword by the routine is preserved across all calls to the routine for a particular name association. It is zero again on the first call for a new name association.

   A name association occurs:

   - Explicitly when dyadic ⎕*NA* is invoked
   - Implicitly on first use of an external routine after it is brought into the active workspace by)*LOAD* or)*COPY*

- Implicitly during)*IN* of an external routine.

Multiple name associations can exist concurrently for one external routine due to shadowing or use of surrogate names. In these cases, the system maintains separate instances of the doubleword values for each association.

4. The first parameter of a VP service parameter list. This is provided so that the external routine can pull itself up by its bootstraps. This is needed because all service requests have parameters lists. Parameter lists need storage, and getting storage for routine execution is a service request.
5. A fullword in which you can place the number of bytes required for routine execution. Then point register 1 at the fourth word of the parameter list (not the fourth parameter value) and call the service routine.
6. A fullword that points to the requested storage area on return from the VP service routine call discussed above. (See "VP: Get Process Storage" on page 97 for more details.)
7. APL2 PerTerm Header
8. Processor 11's local work area for the routine's invocation.

ECV Fields

ECVRQ indicates the type of entry to the external routine. For an external call request it contains 0 (**ECVRQCAL**). The following fields are also defined in this case:

ECVXCDRL Address of the left argument CDR, or 0 if no left argument CDR. This is provided as directed by the **:LARG** field in the NAMES file.

ECVXCDRR Address of the right argument CDR, or 0 if no right argument CDR. This is provided as directed by the **:RARG** field in the NAMES file.

ECVXTLA[1] Left argument object token, or 0 if no left argument.

ECVXTLF[1] Left operand object token, or 0 if no left operand.

ECVXTOP[1] Normally 0, or the object token of the external routine when used as an operator.

ECVXTRF[1] Function or right operand object token or 0. Normally this is the object token of the external routine when it is invoked as a function.

ECVXTRA[1] Right argument object token, or 0 if no right argument.

---

1 This object token can be passed to the **XB**, **XE**, or **XF** services (see Chapter 14, "Code X__: External Call Services" on page 101) but not to the **XC** or **XD** services.

ECVXRLOC Relocation count. This indicates the number of times that the interpreter has reorganized storage in the workspace. When storage is reorganized, object tokens are still valid, but addresses that point into the workspace are not. External routines that retain workspace area addresses across calls from the interpreter, or calls to the interpreter, must also retain the relocation count as it was when the addresses were gotten. If the count changes, they must discard all workspace addresses, and obtain them again from the corresponding object tokens using the **XC** or **XD** services described in Chapter 14, "Code X__: External Call Services" on page 101.

:LINK.FUNCTION Routine Exit: External Call

On completion of an external call request, the following fields may be set:

ECVXCET Result event type. The value provided here will be placed in $\square ET$. This should be set to `0 0` for successful execution. Any other value will cause the APL language processor to signal an error. Note that service requests invoked by the external routine also set this field. Thus the `:LINK.FUNCTION` routine must normally set this field immediately before returning.

If the value provided for ECVXCET is not a standard $\square ET$ code, then a non-zero value must be provided in this field.

ECVXCDRZ Any form object token or 0 if there is no result. Note that this field is ignored if the result event type is set to anything other than `0 0`.

ECVXERMS Address of, or token for, a CDR representing a $\square ES$ message; or 0 to use the system default. Note that this field is ignored if the result event type is set to `0 0`.

The Delete Linkage Call and Exit

This call is made to the `:LINK.FUNCTION` routine whenever the association is broken by the workspace. This can occur because the workspace is terminated by $)CLEAR$, $)LOAD$, $)OFF$, or $)CONTINUE$; because the name has been replaced by $)COPY$ or $)MCOPY$; or because the name has been deleted by $\square EX$, $)ERASE$, or exit from a function that had localized the name and created the association.

The parameters passed to the external routine on this call are similar to those defined in ":LINK.FUNCTION Routine Entry: External Call" on page 29, except that:

- The third parameter will never have been zeroed by APL2, since this is never the first call to the external routine.
- Only the basic section of the ECV is provided, except for one additional field, ECVDET, in which a $\square ET$ code may be returned.

ECVRQ indicates the type of entry to the external routine. For a delete linkage request it contains **ECVRQDEL** (2).

The only result that can be returned is a $\square ET$ signal. Only APL2 language defined $\square ET$ codes may be used. This will be honored by the interpreter during $\square EX$, $)ERASE$, exit from the function that localized the name, and $)IN$ or $)COPY$ that replaces the name. It will be ignored during $)RESET$, $)CLEAR$ and $)LOAD$. In these cases the external routine will be called with ECVISBAD to indicate that the linkage is being dropped unconditionally.

Note: None of the services described in Chapter 14, "Code X__: External Call Services" on page 101 are permitted from within a delete linkage call.

No delete link call is made to environment programs.

Environment Programs

Environment programs share the same entry and exit conditions as non-environment routines. However, there are two important points to notice.

Although environment programs can be either :LINK.OBJECT or :LINK.FUNCTION, no delete linkage call is made to :LINK.FUNCTION environment routines.

When an environment routine passes control to APL2 through AP2TNL or AP2VNL, the state of the registers is stored. The state of registers is restored to this state before each entry to any subroutines that use the environment and before return to the environment program.

Self-Describing External Routine Identification Signature

Processor 11 detects that a routine is self-describing by determining that the routine begins with a routine description signature. Figure 11 shows the contents of a routine description csect.

```
* Routine Description Signature
RDSIG    DS    0F
         L     R15,RDRA-RDSIG(,R15) Address of code
         BR    R15                  Enter code
         DC    H'-1'
         DC    CL4'SIM '            Eyecatcher
RDSIGL   EQU   *-RDSIG              Length of invariant signature
         DC    A(0)                 - Reserved
RDN      DC    CL8'rdname  '        Name of Routine Description
RDRN     DC    CL8'routine '        Name of routine
RDRA     DC    V(routine)           Address of routine
RDDESA   DC    A(RDDES)             Address of description
RDDESEA  DC    A(RDDESE)            Address of end of description
         DC    A(0)                 - Reserved
RDDES    DS    0F                   Start of descriptive information
         DC    CLx'tags'            Routine Description
RDDESE   DS    0F                   Start of descriptive information
* Routine's code follows
```

Figure 11. Routine Description Signature Contents

Where:

`rdname` is the name of the routine description that is used either in a routine list or the right argument of ⎕*NA*.

`routine` is the name of the routine to which the description applies.

`tags` is the actual description composed of processor 11 tags and values.

Note: The APL2 function, `BUILDRD` builds this routine description. For more information, see *APL2 Programming: Using the Supplied Routines*, SH21-1056.

Routine List Identification Signature

Processor 11 detects that a module contains multiple entry points by determining that the module begins with a routine list signature. Figure 12 shows the contents of a routine list csect.

```
RL      CSECT
*
* The next 2 instructions identify this as a routine list
*
        XR    15,15
        BCTR  15,14         RETURN WITH -1
        DC    A(RLTOP)      Top of routine list
        DC    A(16)         Width of routine list
        DC    A(RLLAST)     Last entry in the list
*
* Fill in this list with the names of your routines
*
RLTOP   DC    CL8'objname1',V(routine1),A(0)
        DC    CL8'objname2',V(routine2),A(0)
        DC    CL8'objname3',V(intface1),A(routine3)
        DC    CL8'objname4',V(routine4),A(0)
        DC    CL8'objname5',V(intface2),A(routine5)
*
*             .... additional entries here ....
*
RLLAST  DC    CL8'objnameN',V(routineN),A(0)
        END   RL
```

Figure 12. Routine List Signature contents

Where:

`objname1-objnameN` are the names by which the routines are referenced in the right argument of □*NA*.

`routine1-routineN` are the names of the non-APL routines (or the routine descriptions that describe them.)

`intface1-intface2` are the names of interface management routines. When processor 11 calls an interface management routine, the address of the subsequent routine to be called is placed in register 0. In the example, the subsequent routines are `routine3` and `routine5`.

Note: The APL2 function, `BUILDRL`, builds the routine list. For more information, see *APL2 Programming: Using the Supplied Routines, SH21-1056*.

Chapter 5. General Protocol for Service Calls

APL2 provides services for global auxiliary processors, for local auxiliary processors, and for processor 11 external routines.

For global auxiliary processors,

- Only SVP services are provided (Chapter 11, “Code SC: Shared Variable Services” on page 69).
- The service routine address is provided by an initialization routine, as described in “Global Auxiliary Processor Entry and Exit” on page 24.

For local auxiliary processors and external routines,

- All of the services in the following chapters are provided, except for a few that are noted as limited to only auxiliary processors or only external routines.
- The service routine address is provided when the processor is first entered, as described in Chapter 4, “Entry and Exit Conditions” on page 22.

The service routine entry point address provided at processor entry or initialization is used for all service calls. The register conventions for service calls are:

- Registers 2-13 may have any values, and are restored on completion of the service. (Note that a register 13 save area is ***not*** required.)
- Register 14 must contain the return address in the processor. It will also be restored on service completion.
- Register 15 must contain the service routine entry point. On completion, this will be replaced by a return code set by the service. (The R15 code will also be placed in a return code parameter for most requests.)
- Register 0 will be destroyed by the service routine.
- Register 1 must point to a standard parameter list, pointing to two or more parameters as shown in Figure 13. The high-order bit must be set in the last word of the parameter list. If you are using the operating system **CALL** macro, this is done by including the **VL** parameter. Register 1 will be restored on service completion.

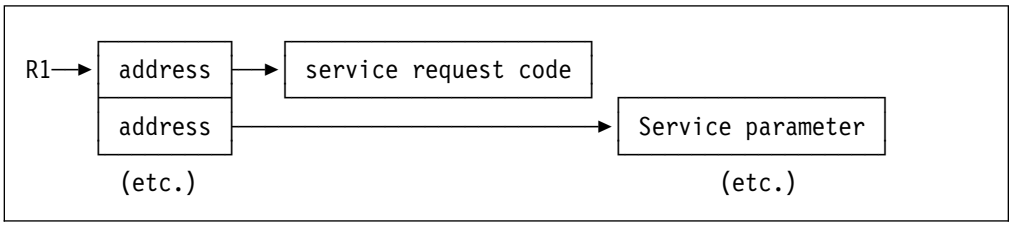


Figure 13. Service Call Parameter List

For the rest of the manual, this interface will be indicated as follows:

```
CALL  (15),(=C'id',parm...),
      VL,MF=(E,listarea)
```

This assumes that you are coding in assembler language using the standard **CALL** macro, have loaded register 15 with the service routine address, and have provided a parameter list area (*listarea*) containing one word for each parameter. The separate *listarea* is required since we assume you are writing reentrant routines.

You may, of course, use other macros or hand-coded calls, or code your routines in other languages, so long as you meet the interfaces defined here.

In C/370 a call to an APL service would have the following format:

```
(*ServRtn) ("XC",&Token,&AddrArea)
```

Where:

`ServRtn` is the address of the APL service routine which is passed as a parameter.

`Token` is a token to be converted to an address

`AddrArea` is an area into which the address will be placed.

In PL/I the same call would have the following format:

```
Call ServRtn('XC',Token,AddrArea)
```

Most of the services provide for return codes to indicate any errors they detect. If, however, a service request code is invalid, or if no service parameter is provided in the parameter list, the task will be ABENDed. Other errors in the service requests or the environment may also cause unexpected ABENDs.

The following chapters are organized by the service request code passed as the first parameter. This is a two-character EBCDIC string whose first character identifies a service class, and second character specifies a service or set of services within that class. The service request codes supported are described in the following chapters:

For a complete list of service request codes, see the Contents on page iii.

Chapter 6. Code D__: Data Conversion Services

The service request codes described in this chapter are:

DE Translate VS APL Zcode to EBCDIC
DN Numeric data type conversion
DU Translate with caller translate table
DX Convert extended character sets
DZ Translate EBCDIC to VS APL Zcode

DE: Translate from VS APL Zcode to EBCDIC

This service operates on character data of arbitrary length which is in the character set used by VS APL, converting it to the standard EBCDIC character set used by APL2. This would most likely come into use if an auxiliary processor is accessing a file that contains VS APL data.

```
CALL  (15),(=C'DE',retcode,datalen,outbuff,data),
      VL,MF=(E,listarea)
```

Set by processor

datalen A fullword containing the length of the data to be translated.

Note: The system will assume that *outbuff* is at least that long.

data The data (of length *datalen*) to be converted.

Returned to processor

retcode A fullword service completion code. The following return codes are defined:

0 normal completion
101 invalid parameters

outbuff The translated string, of length *datalen*.

DN: Change Data Format of One or More Numbers

Produces a list of numbers in the output area, in the format specified by the output type. The input area is analyzed according to the input type. Processing then begins with the n^{th} number (specified by element index) of the (ravelled) input array, and continues for the specified number of elements.

CALL (15),(=C'DN',*retcode,bufflen,outbuff,data,types,index,count[,descrip]***),**
VL,MF=(E,*listarea***)**

Set by processor

bufflen A fullword containing the length of the output buffer.

data The data to be converted. Its format depends on the first element of *types*.

types Two two-byte fields, each containing a two-character data type code. The first field determines the format of *data*, while the second determines the format of *outbuff*. The possible character values in each field are:

A0 APL object
B1 Boolean (1 bit, packed 8 per byte)
B8 8-bit binary (unsigned)
I2 halfword binary
I4 fullword binary
E4 1-word floating point
E8 2-word floating point
EX 4-word floating point
C0 An item from a CDR

Note: Input type **A0** may be in either VS APL or CDR format. *Data* is the VS APL descriptor word or CDR header respectively. Output type **A0** is always produced in CDR format. *outbuf* will contain a complete CDR in dense form.

Note: Type **C0**, which may only be used for input, deals with an *item* from a CDR. *data* is the data array itself, not its CDR header or descriptor. *descrip* is the simple descriptor for that data array.

index A fullword containing an origin-0 index into the input data. This indicates the first element to be converted.

count A fullword containing the count of elements to be converted.

descrip This parameter is used only with input type **C0**. It is the simple (never G-type) CDR descriptor of the *data* array. (See Chapter 3, “APL2 Data Representation” on page 13 for details on the format of a CDR.)

Returned to processor

retcode A fullword service completion code. The following return codes are defined:

0 Normal completion
13 Data element could not be converted
16 APL input has too few elements
17 Output area is too small
99 Invalid or unsupported request

outbuff The numeric results. The format of the results depends on the second element of *types*.

DU: Translate with Caller Supplied Table

This service operates on 1-byte character data of arbitrary length, translating it under control of a translate table provided by the processor.

```
CALL  (15),(=C'DU',retcode,datalen,outbuff,data,trantab),
      VL,MF=(E,listarea)
```

Set by processor

datalen A fullword containing the length of the data to be translated.

Note: The system will assume that *outbuff* is at least that long.

data The data (of length *datalen*) to be converted.

trantab A 256-byte translate table which will control the translation as defined by the hardware TR (translate) instruction, except that the length of the data to be translated is not limited to 256, and the condition code may have been changed.

Returned to processor

retcode A fullword service completion code. The following return codes are defined:

0 normal completion
101 invalid parameters

outbuff The translated string, of length *datalen*.

DX: Convert Extended Character Data

This service is provided as an aid in manipulating extended character sets such as Japanese Kanji, which contain more than 256 characters.

Outside of APL, such character sets are normally represented using a double byte character set (DBCS). “Shift-out” and “Shift-in” characters (X'0E' and X'0F') are often used to switch between the EBCDIC single byte character set (SBCS) and the DBCS.

APL supports a 4-byte character representation which permits a large number of character sets to be represented concurrently. The first two byes of each 4-byte character are a character set code. Character set X'0000' is defined to be EBCDIC/APL.

CALL (15),(=C'DX',*retcode,outlen,outbuff,data,datalen,options[,charcode]***),**
VL,MF=(E,*listarea***)**

Set by processor

outlen A fullword containing the length of *outbuff*.

data The data to be converted. Its format depends on *options*, while its length is specified by *datalen*.

datalen A fullword containing the length (in bytes) of *data*.

options A two-character field containing one of the following, left aligned:

- **U** Pack 4-byte data into simple 1-byte data if possible, else into simple 2-byte data without SO/SI characters.
- **M** Pack 4-byte data into mixed SBCS/DBCS, starting and ending in 1-byte mode.
- **W1** Unpack data that starts in 1-byte mode into APL characters, leaving it unchanged if no Shift-out is found, else creating extended (4-byte) characters.
- **W2** Unpack double-byte data into 4-byte data. All characters are valid for this conversion.

  Note: For options **U** and **M**, *outbuff* may identify the same storage as *data*, but for the other two options, they must be distinct areas. Overlaps will usually result in program exceptions.

charcode An optional halfword specifying a character set code to use (on the “unpack” operations), or to check against (on the “pack” operations). If this parameter is not provided, the current session character set is used, as defined by the **DBCS** invocation option. There is a special **DBCS(OFF)** mode with limited validation. This mode may be requested here by supplying **H'-1'**.

Returned to processor

retcode A fullword service completion code. The following return codes are defined:

- 0 Normal completion, no DBCS characters found

1 Normal completion, DBCS characters found
2 Invalid characters encountered
3 Output area too small

outlen The length (in bytes) required by the result data.

outbuff The buffer containing the converted data. The format of the data depends on *options*.

Special Notes on Character Data Conversion

This service has four basic variants controlled by the two character *options* defined above. There are a number of other features that are not immediately apparent:

1. If the output area length is specified as zero, the service will scan the input data to determine how much space it would need if it were converted according to the specified option. That size will be returned in the *outlen* parameter.
2. If character set code -1 is specified for the **U** or **M** options, any character set code is valid in extended input data.
3. If the character set code is non-negative for the **U** or **M** options, a character set code mismatch produces return code 2. Invalid data is mapped as X'0000'.
4. Return code 1 is especially useful when options **U** or **W1** are requested, since it indicates which conversion was performed.
5. Using option **W1**, return code 2 may occur for either of the following reasons:
   - A shift-out or shift-in byte was found at a point where it could not logically cause a shift.
   - A DBCS character had a X'00' in its high byte.
6. Using options **U** or **M**, return code 2 may occur for any of the following reasons:
   - A shift-out or shift-in byte was found in the input data.
   - An APL character had a nonzero character set code.
   - A DBCS character had a character set code which did not match the nonzero code which was specified or defaulted.
7. When return code 2 is issued for option *U*, the data is always returned in 2-byte form.
8. In all cases, when an invalid character is encountered, it is converted to X'0000' and processing continues.

IBM standards for DBCS character sets impose the following limitations:

1. Neither byte of the two-byte DBCS character may occupy the first EBCDIC quadrant (X'00' through X'3F').
2. Neither byte of the two-byte DBCS character may be X'FF'.
3. Neither byte of the two-byte DBCS character may be X'40' unless the other byte is also X'40'.

This service compares all DBCS characters to those standards, and returns code 2 (converting the offending characters to X'0000') for all requests except **W2**.

DZ: Translate from EBCDIC to VS APL Zcode

This service operates on standard (1-byte) character data of arbitrary length, converting it to the character set used by VS APL. This would most likely come into use if an auxiliary processor is writing data into an existing file which already contains VS APL data.

```
CALL  (15),(=C'DZ',retcode,datalen,outbuff,data),
      VL,MF=(E,listarea)
```

Set by processor

datalen A fullword containing the length of the data to be translated.

Note: The system will assume that *outbuff* is at least that long.

data The data (of length *datalen*) to be converted.

Returned to processor

retcode A fullword service completion code. The following return codes are defined:

0 normal completion
101 invalid parameters

outbuff The translated string, of length *datalen*.

Chapter 7. Code E__: Error Handling Services

The service request codes described in this chapter are:

ED Take a dump and continue
ET Terminate abnormally
EX Set or clear an ABEND exit
EZ Designate a permanent routine

ED: Produce a Dump

```
CALL  (15),(=C'ED',dumpid[,psw[,regs]]),
      VL,MF=(E,listarea)
```

Set by processor

dumpid Four character dump identifier. Any alphanumeric string may be used, but the form *nnnc* is recommended for APs, where *nnn* is the AP number and *c* may be any alphabetic character.

psw An optional eight-byte Program Status Word associated with the problem.

regs An optional 16 word area containing register values associated with the problem.

Returned to processor: None.

Note: If a dump is requested from within an **EX** abend exit entered because of a program check, the *psw* and *regs* parameters are ignored, and the PSW and registers associated with the program check are automatically included in the dump.

Description: A number of areas are dumped automatically:

The perterm and global table
The first 4K of the workspace
(PSW address)-256 for 1024 bytes
(each register)-256 for 1024 bytes

The following areas are also dumped if **DEBUG(4)** is set:

The full workspace
Shared memory
The executor module
The interpreter module
All global storage
All task storage for the current task

ET: Request Abnormal Termination

```
CALL  (15),(=C'ET',abendcode),
      VL,MF=(E,listarea)
```

Set by processor

abendcode A fullword containing a number between 1 and 999. (Numbers larger than 999 are reserved for IBM use.)

Returned to processor: None.

Note: If an **EX** exit currently exists for the process requesting the ABEND, that exit routine will gain control. You may want to clear the exit using the **EX** service before issuing **ET**.

EX: Set or Clear an ABEND Exit

Note: **EX** cannot be used by routines whose routine description contains an `:INIT` tag.

This service specifies the address of an exit routine which will be given control if an ABEND or program check occurs while the process is in control. The exit routine is not given control on attention signals unless the process is terminated because of repeated unacknowledged signals.

Any previous exit for the same process is cleared when an exit is set. The service itself provides no direct support for stacking exits, but the previous exit address will be returned if requested. This allows applications to restore a previous exit at a later time.

The abend exit *will* be given control even on nonretryable abends for which APL2 gains control. On an MVS system these include operator cancel, timeout, etc. In general, VM does not give APL2 control in nonretryable situations.

CALL (15),(=C'EX',*exitroutine*|0 *[,oldexit]***),**
VL,MF=(E,*listarea***)**

Set by processor

exitroutine A fullword containing the address of the routine to be given control. See "Entry/exit conditions for abend exits." Call this service with binary zeros in the field to remove the abend exit for this process.

Returned to processor

oldexit An optional fullword in which the address of any previously defined exit routine will be returned. Zero will be returned if no exit was previously in effect.

Entry/exit conditions for abend exits

When the abend exit is entered:

- Register 13 points to a standard save area.
- Register 14 contains a return address which *must* be used to return from the abend exit.
- Register 15 contains the address of the exit routine.
- Register 1 points to a parameter list (see Figure 14 on page 47).
- Registers 2 through 12 contain service routine information which must be restored before returning from the abend exit.
- Extended addressing mode (31 bit mode) will be in effect where available, regardless of the mode the processor may have been using.

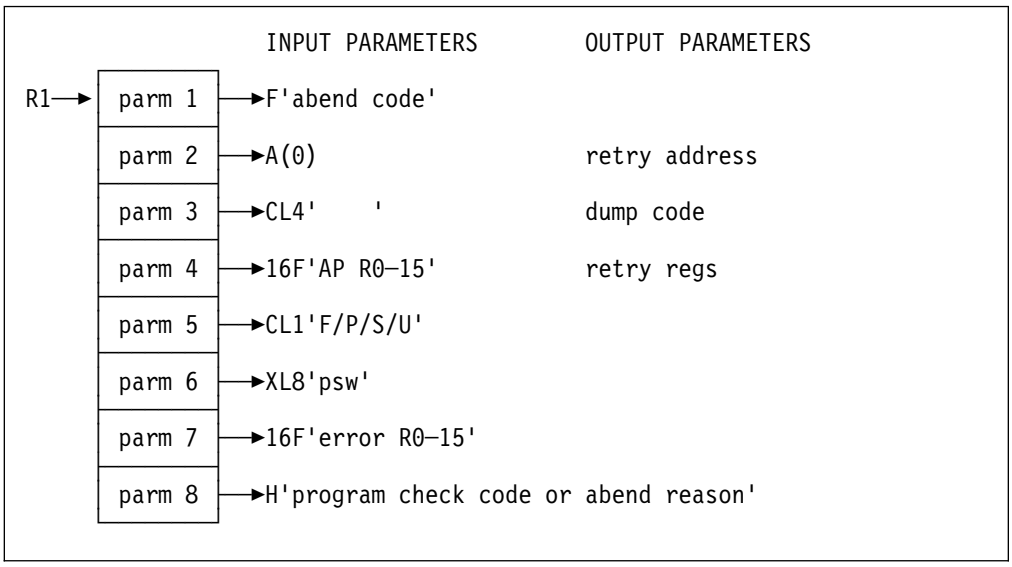


Figure 14. Abend Exit Routine Parameter List

parm 1 The abend code is in the form 00*sssuuu*, where *sss* is a system abend code or *uuu* is a user abend code.

parm 2 A retry address may optionally be supplied before returning. If it is not, the process will be terminated on return from the abend exit. If a retry address is supplied, execution will be resumed there after completing error recovery.

Note: On systems that support extended addressing, the retry routine will be entered in the addressing mode that the *program* was using immediately before the error. Your program may have used a system-assisted linkage to call another program, and the error may have occurred in that program. But neither the addressing mode of that program, nor the high-order bit of the retry address you provide, affects the mode in which your retry routine is entered. APL2 does, however, attempt to clean up your retry address by zeroing bits 1-7 if bit 0 is not on. Bits 1-7 *must* be zero if the program will be entered in 24-bit mode; otherwise an immediate specification exception will occur. On the other hand, if you intend to retry to an address above 16 megabytes, you *must* set bit zero on to keep APL2 from destroying your address.

parm 3 A dump code may optionally be supplied. If it is, and if dumps are supported by the APL2 session (via **APLDUMP**), a dump will be taken on return from the abend exit. The dump code is a 4-character alphanumeric string. The form *nnnc* is recommended for APs, where *nnn* is the AP number and *c* may be any alphabetic character.

parm 4 On entry to the exit routine, this parameter is a set of registers as of the last service call issued by the processor. This will typically be useful in reestablishing addressability. The registers may be modified as desired by the exit routine. The updated register set will be used (except R14) when retrying.

parm 5 The type of abend is one of:

- **F** Force off (nonretryable) system or subsystem initiated abend.
- **P** Program check. PSW and regs are available.
- **S** System or subsystem initiated abend.
- **U** User (processor or APL2) initiated abend.

The following two parameters are provided only for program checks.

parm 6 This is the hardware Program Status Word (PSW) at the time of the error; except that for CMS, if the program check occurred within a supervisor service (SVC routine), the PSW will indicate the point at which the SVC was issued.

parm 7 These are the registers that correspond to the PSW in parm 6.

> **Note:** The program cannot assume that the PSW and registers are directly related to the work it was doing. The error may have occurred in some program it has called. Do not attempt to use this set of registers to restore addressability.

Before entering the abend exit routine, the exit itself is disabled. If the exit routine should abend or program check, the entire process will be terminated. A dump will be taken which reflects the most recent abend.

After the abend exit routine returns, it is automatically enabled. For this reason, it is usually best to do most cleanup or recovery after retrying. However, for program checks, the PSW and error register information is available *only* during the abend exit.

The program should be carefully written to avoid abend loops during cleanup. One recommended technique is to supply a retry address in a process control block, and zero it in the abend exit. The cleanup routine can then, for each step:

1. Store the address of step n+1 for retry
2. Execute step n

EZ: Designate a Permanent Routine

Note: EZ applies only to processor 11 external routines.

```
CALL  (15),(=C'EZ',[C'S'|C'C'],retcode,entryadd,token),
      VL,MF=(E,listarea)
```

Set by processor

C'S' or C'C' For Set or Clear

entryadd A fullword address of the code to be entered at APL2 termination

token A fullword token provided to the routine when it is entered during APL2 termination.

Returned to processor

retcode A fullword service completion code. See Figure 15.

Figure 15. EZ Service Return Codes

Code	Definition
0	Success
4	Routine entry point not found (clear request)
8	Insufficient space
12	Invalid subcode (must be 'S' or 'C')

The 'EZ' service specifies a routine that is to stay active across replacement of the workspace. The 'EZ' service allows an external routine to nominate an entry point that will be entered when APL2 is shut down. Since processor 11 deletes all active external routines when the workspace is replaced (`)CLEAR`, `)LOAD`, `)CONTINUE`, or `)OFF`) such routines must take special action to ensure that the specified entry point is still available at APL termination. This can be done by loading the necessary code as a CMS nucleus extension or by issuing a LOAD (SVC 8) request for it. If the routine was loaded as a CMS nucleus extension, it must be explicitly deleted with a NUCXDROP command.

When APL2 terminates, each entry point nominated via an 'EZ' request is entered with the following (standard OS) linkage:

```
R1 => A(token+X'8000000')
R13=> 18 word OS save area
R14=  return address
R15=  entry point address
```

Chapter 8. Code F__: File System Services

The file system provided through these services corresponds to that used by AP 121 and processor 12. In particular, sequential and direct files are supported. See the description of AP 121 in *APL2 Programming: System Services Reference* for limitations on these file types.

Consult the material on APL2 Libraries, Workspaces, and Data Files in that same manual for information about the library structure under CMS and TSO. Note in particular that the services described here apply only to individual logical files within a library, not to the library as a whole.

The service request codes described in this chapter are:

FA Open an APL file
FC Create an APL file
FD Delete an APL file
FG Access a file in a file group
FL List APL files
FR Read a record from an open APL file
FS Change the size of an APL file
FW Write a record to an open APL file
FZ Close an APL file

File Services Return Codes

0	Normal completion
1	No record found
2	Maximum records or extents
3	Non-EBCDIC/APL characters
4	Invalid APL object in file
5	End of file
6	File type and access conflict
7	Record length error
9	APL library is full
10	User quota exceeded or attention hit
11	Total data does not fit within the file size limit
14	VSAM file not open
15	APL data file not open
23	File must be opened for direct
24	Improper library reference
25	File already exists
26	File does not exist
28	File in use by others
29	Illegal change to file not owned
30	File protected from operation
31	File open limit exceeded
32	Getmain for I/O buffer failed
34	Incorrect or missing password, or you are not authorized to write to this file.
35	I/O error, unknown error, or invalid library
36	This is not an APL file
37	OK, but file size has been reduced
38	Invalid record length

Figure 16. Return Codes for APL File Services

Note: These return codes do not apply to **FG**.

FA: Open an APL File

CALL (15),(=C'FA',*retcode,libno,fname,pass,token,use[,maxlen[,records]]***),**
VL,MF=(E,*listarea***)**

Set by processor

libno A fullword containing the number of the library within which the file exists. The library number may be set to zero to indicate that the file is in the user's private library.

fname An 8-character field containing the name of the file to be opened, padded with blanks.

pass An 8-character field containing an optional password for the library. If a password is required, but none is supplied here, the user will be prompted. The field must be set to blanks if no password is desired.

use A 2-character field in which the first character is **R** for read access or **W** for write access, and the second one contains an **S** for sequential processing or **D** for direct access. When the second character is **D**, read is also allowed during write access.

Note: A file created with the **D** attribute can be opened for either **D** or **S** processing, but a file created with the **S** attribute can be opened only for **S** processing.

Returned to processor

retcode A fullword service completion code. See Figure 16 on page 51.

token A fullword *file token*. This value must be provided on subsequent **FR** and **FW** requests for the file, and must be "turned in" on the **FZ** request that closes the file.

maxlen An optional fullword in which the service returns the maximum length (in bytes) that any record in the file can ever use. This value is 0 for sequential files and for direct files with arbitrary data length.

records An optional fullword in which the service returns the number of records that currently exist in the file.

FC: Create an APL File

```
CALL  (15),(=C'FC',retcode,libno,fname,pass,maxsize,usetype,[maxlen]),
      VL,MF=(E,listarea)
```

Set by processor

libno A fullword containing the number of the library within which the file should be created. The library number can be set to zero to indicate that the file is to be allocated in the user's private library.

fname An 8-character field containing the name of the file to be created, padded with blanks.

pass An 8-character field containing an optional password for the library. If a password is required, but none is supplied here, the user will be prompted. The field must be set to blanks if no password is desired.

maxsize A fullword containing the maximum size of the file in bytes. If this field is set to zero, the size of the file will be limited only by available space in the library.

usetype A 2-character field in which the first byte is ignored, but the second must contain an **S** or **D** to indicate whether a Sequential or Direct file is being created. See “FA: Open an APL File” on page 52 for further discussion of this.

maxlen A fullword containing the maximum length (in bytes) that any record in the file will ever require. This value is ignored for sequential files. For direct files it includes the four-byte record length field, and must be between 0 and 4074. A length of 0 indicates that the file should allow arbitrarily sized objects.

Returned to processor

retcode A fullword service completion code. See Figure 16 on page 51.

FD: Delete an APL File

```
CALL  (15),(=C'FD',retcode,libno,fname,pass),
      VL,MF=(E,listarea)
```

Set by processor

libno A fullword containing the number of the library that contains the file to be deleted. The library number may be set to zero to indicate that the file is in the user's private library.

fname An 8-character field containing the name of the file being deleted, padded with blanks.

pass An 8-character field containing an optional password for the library. If a password is required, but none is supplied here, the user will be prompted. The field must be set to blanks if no password is desired.

Returned to processor

retcode A fullword service completion code. See Figure 16 on page 51.

Note: The file cannot be deleted if currently open. See “FZ: Close an APL File” on page 61.

FG: Access a File in a File Group

This service provides sequential read access to existing system files that can be located using two levels of name qualification, called a *group name* and a *file name*.

Under CMS the "group name" is a CMS **file type**, and the "file name" is a CMS **file name**. The CMS file mode is *, that is, all accessed disks are searched in the standard order, and the first file found of the correct name is used.

Under TSO the "group name" is a **ddname**, which must have been allocated to point to a partitioned data set. The "file name" is a member name within the PDS. Standard PDS concatenation is supported, and the first file found with the proper member name will be used.

The following call is used to open a file in the group:

```
CALL  (15),(=C'FG',retcode,C'OS',token,gname,fname),
      VL,MF=(E,listarea)
```

Set by processor

token Normally zero for this call. If a token is still available from a previous open for the same group (that is, the file has not been closed), that token can be provided here to request a "jump" from one file to another within the group. This can provide performance advantages under TSO.

gname An 8-character field containing the name of the file group, padded with blanks.

fname An 8-character field containing the name of the individual file within a group, padded with blanks. This is an MVS member name or a CMS file name.

Returned to processor

retcode A fullword service completion code.

- **0** File opened successfully.
- **7** TSO: Fixed record format file without LRECL.
- **24** TSO: ddname not allocated, or not a PDS.
- **26** TSO: *fname* not found as a member in the PDS.
  CMS: File *gname fname* * does not exist.
- **30** TSO: User not authorized to access the file.
- **32** Not enough free storage available.
- **35** TSO: I/O error locating the file.
- **36** TSO: Not a PDS concatenation.[2]

token A fullword integer token is returned here. This token must be provided on subsequent read and close calls.

Note: Files can be left open for as long as needed, but they must eventually be closed by the process that opened them.

[2] This error also occurs if a member name was specified as a part of the data set name when the ddname was defined.

The following call is used to read a record from an open file:

```
CALL  (15),(=C'FG',retcode,C'SR',token,bufflen,buffer,reclen),
      VL,MF=(E,listarea)
```

Set by processor

token A fullword containing the token returned when the file was opened.

bufflen A fullword containing the length of the area pointed to by *buffer*.

Returned to processor

retcode A fullword service completion code.

- **0** Record read successfully.
- **5** End of File.
- **7** Record too long for buffer provided.
- **26** CMS: File *gname fname* * does not exist.
- **35** I/O error reading the record.
- **36** Not a qualified name file
  TSO: May also mean size of a block read is not multiple of LRECL
- **50** Invalid request for this system
- **60** TSO: Invalid record format

buffer A record is returned in this area.

recln A fullword integer field into which the length of the returned record will be placed by the service.

Warning: Abends might occur under TSO if a request is made using a token that was obtained from opening a file by another task.

The following call is used to close a previously opened file:

```
CALL  (15),(=C'FG',retcode,C'CL',token),
      VL,MF=(E,listarea)
```

Set by processor

token A fullword containing the token returned when the file was opened.

Returned to processor

retcode A fullword service completion code.

- **0** File was successfully closed.
- **35** TSO: I/O error closing the file.

token This field is reset to zero. The token must not be used again regardless of the return code.

FL: List APL Files

CALL (15),(=C'FL',*retcode,libno,,pass,bufflen,buffer*[**,***start*[**,***end*]]**),**
VL,MF=(E,*listarea***)**

Set by processor

libno A fullword field containing the APL library number.

pass An 8-character field containing an optional password for the library. If a password is required but none is supplied here, you are prompted. The field must be set to blanks if no password is desired.

bufflen A fullword containing the length of the area pointed to by the *buffer* parameter.

start An 8-character field containing an optional starting delimiter for the search. If not provided, the default is '*A* '.

end An 8-character field containing an optional ending delimiter for the list. If not provided, the default is to search to the end of the list.

Returned to processor

retcode A fullword service completion code. See Figure 16 on page 51.

bufflen The number of bytes actually used in *buffer*.

buffer The list of names. Each name is padded to 8 characters with blanks, and each is followed by one blank.

start If *retcode* is 7 (buffer too small), and this parameter was provided, it contains the next name that would have been added to the list. If the application wants to avoid rebuilding the entire list, it can issue a second call with this start parameter to obtain the next segment of the list.

FR: Read an APL File Record

```
CALL  (15),(=C'FR',retcode,bufflen,buffer,recno,token),
      VL,MF=(E,listarea)
```

Set by processor

bufflen A fullword containing the length of the area pointed to by the *buffer* parameter. For optimum performance, the following buffer sizes are recommended:

Sequential files: At least one page (4096 bytes)
Direct files with fixed size: At least “maxlen” bytes
Direct files with arbitrary size: At least 1 page (4096 bytes).

recno A fullword containing the relative record number in the file if the file was opened for direct processing (see the **FA** service). Its contents are ignored if the file was opened for sequential processing.

token A fullword containing the *file token* provided when the file was opened.

Returned to processor

retcode A fullword service completion code. See Figure 16 on page 51.

buffer The length of the record, returned as the first four bytes of the buffer, followed by the data. The length field is returned even if the data does not fit (return code 7). (The length is in bytes, and does not include itself.) If *retcode* is 7 (buffer too small), the first four bytes contain the buffer size required to read the record.

recno A fullword containing the relative record number in the file. This is the number associated with the record in *buffer*.

token If an error occurs, the file may be closed. If that happens, this field will be zeroed to indicate that the file is closed and the *token* is no longer available.

FS: Change the Size of an APL File

```
CALL  (15),(=C'FS',retcode,libno,fname,pass,maxsize),
      VL,MF=(E,listarea)
```

Set by processor

libno A fullword containing the number of the library within which the file exists. The library number may be set to zero to indicate that the file is in the user's private library.

fname An 8-character field containing the name of the file, padded with blanks.

pass An 8-character field containing an optional password for the library. If a password is required, but none is supplied here, the user will be prompted. The field must be set to blanks if no password is desired.

maxsize A fullword containing the new maximum size of the file in bytes. If this field is set to zero, the size of the file will be limited only by available space in the library.

Returned to processor

retcode A fullword service completion code. See Figure 16 on page 51.

Note: The file size cannot be changed if the file is currently open. See “FZ: Close an APL File” on page 61.

FW: Write an APL File Record

```
CALL  (15),(=C'FW',retcode,,buffer,recno,token),
      VL,MF=(E,listarea)
```

Set by processor

buffer A buffer that contains the record to be written. The length of the record must be provided as the first four bytes of the buffer, followed by the data. (The length is in bytes, and does not include itself.)

recno A fullword containing the relative record number in the file if the file was opened for direct processing (see the **FA** service). In this case the record must replace one that already exists in the file. The contents of this field are ignored if the file was opened for sequential processing.

token A fullword containing the *file token* provided when the file was opened.

Returned to processor

retcode A fullword service completion code. See Figure 16 on page 51.

recno A fullword containing the relative record number in the file of the record that was just written.

token If an error occurs, the file may be closed. If that happens, this field will be zeroed to indicate that the file is closed and the *token* is no longer available.

Note: A write request may be issued only if the file was opened for write processing.

For files created as type **D** (regardless of how they are opened), the record length cannot exceed that which was specified as a maximum when the file was created.

FZ: Close an APL File

```
CALL  (15),(=C'FZ',retcode,,,,token),
      VL,MF=(E,listarea)
```

Set by processor

token A fullword containing the *file token* provided when the file was opened. This service “turns in” that token, that is, it must not be used later for any other file requests.

Returned to processor

retcode A fullword service completion code. See Figure 16 on page 51.

token This field will be zeroed to indicate that the *token* is no longer available.

Note: This request has three unused parameters so that the file token will be the sixth parameter in all requests.

Chapter 9. Code M__: Message Services

The service request codes described in this chapter are:

MC Check for message existence
MF Format a message (with optional display)

MC: Check for Message Existence

```
CALL  (15),(=C'MC',retcode,msgnum),
      VL,MF=(E,listarea)
```

Set by processor

msgnum A fullword containing the message number to check for.

Returned to processor

retcode A fullword service completion code. The following return codes are defined:

0 the message does exist
1 the message does not exist

The message number “exists” if it can be found in either the standard English table provided as a part of the product or a national language (**APL2LANG**) file which has been made active using $\square NLT$.

Note: The information returned is valid only until the next time $\square NLT$ is set (or unshadowed) within the workspace. Changing the national language may either make new messages available or make previously valid messages inaccessible.

MF: Format a Message

This service gives processors access to the message services used by the APL2 product. Messages may be displayed, queued, or returned to the caller. The current national language table is used, substitution fields are supported, and a message ID is optionally supplied.

The message numbers are defined in either the standard English table provided as a part of the product or a national language (**APL2LANG**) file which has been made active using ⎕*NLT*.

After formatting, the total message length including the message ID must not exceed 255 characters.

CALL (15),(=C'MF',*retcode,msgnum,opt[,outarea,outlen][,fill-val,fill-len]...***),**
VL,MF=(E,*listarea***)**

Set by processor

msgnum A fullword message number which identifies the message to use. Message numbers are defined in the **DEFAULT APL2LANG** file.

Note: It is possible for installations or users to define additional message numbers in **APL2LANG** files, and then to use those numbers from this service while the **APL2LANG** file is in effect for the session. If you intend to use this capability, you should issue the **MC** service first, since an ABEND will occur if the message cannot be found.

opt A one-character code indicating what should be done with the message:

D Display the message along with data displayed by the interpreter as a part of the APL session. If the session manager is being used, it will be included in the session log.

Q Queue the message for display on a subsequent)*MORE* request.

Note: If **DEBUG(1)** is in effect this option will be overridden, and **D** will be used instead.

R Return the formatted message in *outarea*. The returned message will begin with a message ID if **DEBUG(32)** is in effect.

outlen A fullword containing the length of *outarea*.

Note: This parameter (and *outarea*) must be supplied if, and only if, *opt* **R** is specified.

fill-val An optional value to be substituted into the message. Message substitution fields are numbered, beginning with 1 for each message (see the **DEFAULT APL2LANG** file). The first *fill-val* provided here corresponds to substitution field number 1, and so forth. The field must contain character data. (For double-byte languages, mixed DBCS data is permitted.)

If the current message definition contains more substitution fields than are provided for by *fill-val* parameters, the additional fields will be filled with '**'. If extra *fill-val* parameters are provided, they will be ignored.

fill-len A fullword containing the length of the preceding *fill-val*. This length must never exceed 255. If the length is specified as zero, an empty field is substituted into the message. If the length is negative, the field will be treated as unsubstituted, and will display as '**.'.

Note that although *fill-val* parameters are optional, there must be a *fill-len* parameter for each one that is provided. In other words, it is really *fill-val/fill-len* pairs that are optional.

Returned to processor

retcode A fullword service completion code. The following return codes are defined:

0 normal completion
1 required virtual storage is not available
2 the message is too long (or *outarea* is too short)

outarea The formatted message is placed in this area. This parameter (and *outlen*) must be supplied if, and only if, *opt* **R** is specified.

Note: For double-byte languages, the formatted message will be in mixed DBCS format if either the message definition or the substitutions contain any DBCS characters.

outlen A fullword contain the length of the message placed in *outarea*. If *retcode* is **2**, a minimum known requirement will be returned.

Note that the message length may increase and decrease several times during processing. If, after a length error, the **MF** service is reissued using the length returned here, it may take several iterations to determine the actual size required. Also, the final message will typically be smaller than the maximum requirement during processing.

Note: This parameter (and *outarea*) must be supplied if, and only if, *opt* **R** is specified.

Chapter 10. Code P__: Process Services

The following services are available only to Auxiliary Processors.

Process services use one-word blocks called *event control blocks* (**ECB**s) to synchronize the operations of two processes. The internal format and content of an ECB is system dependent, but may be partially controlled by the **POST**ing process. See “CSVSCAN: Scan for an Offer” on page 81 for additional usage of ECBs.

There is no return code from any of the processor services. Information about the success of the operation is often available in an ECB. Invalid parameters cause an ABEND of the processor.

The service request codes described in this chapter are:

PP Post an ECB
PT Start a timer
PW Wait for an event

PP: Post an ECB

Send a signal to another task in the same address space or virtual machine. This signal will terminate an operating system **WAIT** or a **PW** service that has suspended any task on that ECB. It will also set a *post bit* in the ECB so that a later **WAIT** or **PW** will complete immediately.

```
CALL  (15),(=C'PP',ecb,postcode),
      VL,MF=(E,listarea)
```

Set by processor

postcode A fullword containing a nonnegative binary number which cannot exceed 32767. The number will be placed in the low order halfword of the ECB.

Returned to processor

ecb A fullword ECB in the same virtual machine or address space which will have been posted.

PT: Start a Timer

This request sets an “alarm clock” which will send a signal after a specified amount of “wall clock” time has elapsed.

A timer that has not expired is cancelled by a subsequent timer request from the same process, or by the process's termination. (Note that a time interval of zero effectively cancels any outstanding timer.)

```
CALL  (15),(=C'PT',ecb,time),
        VL,MF=(E,listarea)
```

Set by processor

time A fullword containing the length of time, in milliseconds.

Returned to processor

ecb A fullword ECB which will be posted when (or soon after) the time interval has elapsed.

Note: Control returns immediately, although normally the ECB will not yet have been posted. Use the **PW** service to wait for the timer signal.

PW: Wait for an Event

CALL (15),(=C'PW',*[posted,ecb[,ecb]...]***),**
VL,MF=(E,*listarea***)**

Set by processor: None.

Returned to processor

posted A fullword pointer to a posted ECB.

ecb A fullword ECB which has been posted asynchronously by another task.

Description: If no ECBs are listed, this is a "dispatch" request, which gives all other APL2 processes an opportunity to execute (if not in an unposted WAIT) before returning control to the issuing process.

If one or more ECBs are specified, each call returns information about one posted ECB. The second parameter word (not the parameter list entry) will point to the ECB about which information is being returned. The post code (if any) will be right justified in the ECB. The first two bytes of the ECB may contain system dependent information which should be ignored by the processor. If multiple ECBs are specified, control may be returned when any one of them has been posted.

To avoid spurious "posted" indications, all ECBs must be initialized to binary zero before their use. Note that when multiple ECBs are waited on, more than one may have been posted on return. The normal procedure would be for the processor to handle the event pointed to by the second parameter word, clear that ECB, then reissue the wait. At that time a second posted ECB would be noticed, and the processor would be redispatched immediately.

Chapter 11. Code SC: Shared Variable Services

The service request codes described in this chapter are:

SC Shared variable services

This service provides communication and data transfer between auxiliary processors and the SVP. The parameter list itself is very simple, but the second parameter is an often complex parameter block:

```
CALL  (15),(=C'SC',svpblock),
      VL,MF=(E,listarea)
```

svpblock One of three different parameter blocks must be provided here, depending on the type of request being made. In all cases the first halfword of the parameter block identifies the request, and hence the format of the remainder of the block.

The three parameter blocks are associated with three classes of requests:

1. Processor control requests use a *processor control vector* (PCV). These are the first and last requests a processor will use; they establish or break a connection with the shared variable processor.
2. Share control requests use a *share control vector* (SCV). These are the "workhorse" requests; they handle all shared variable connection, status, and data transfer.
3. The data format request uses an *SVP data format* block (SDF). This request permits data compatibility with other APL systems.

Each of these parameter blocks is described by an assembler language mapping macro (AP2PCV, AP2SCV, or AP2SDF) which is distributed with the APL2 product. If you are writing an auxiliary processor in some other language, you will have to provide your own mapping of these parameter blocks.

Another macro, AP2CSVPE, provides declarations of the request code values and return code values. The names defined by that macro will be used throughout this chapter to refer to the individual requests and return codes.

Note: If you need to use the AP2PCV, AP2SCV, AP2SDF or AP2CSVPE mapping macros, see Appendix C, "Macros Intended for Customer Use" on page 182 for the information you need to get from your system programmer.

The requests are:

Processor Control Requests

CSVON	signon
CSVOFF	signoff

Share Control Requests

CSVCOPY	copy
CSVQUERY	query
CSVREF	reference
CSVREL	release
CSVRET	retract
CSVSCAN	scan
CSVSEEAC	see access information
CSVSETAC	set acv
CSVSHARE	share
CSVSPEC	specify
CSVSTATE	state

Data Format Request

CSVDFORM	set data format

SVP Processor Control

Processor services are related to the state of the auxiliary processor itself, without reference to particular shared variables. The two processor services are CSVON (signon) and CSVOFF (signoff). An assembler language mapping of the PCV is distributed with the product as macro AP2PCV. Figure 17 shows its overall layout.

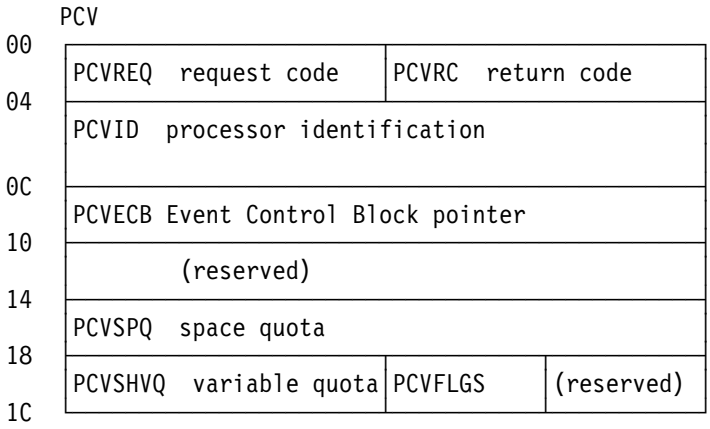


Figure 17. PCV: SVP Processor Request Block

See "CSVON: Signon" and "CSVOFF: Signoff" on page 73 for usage of the fields in this block.

CSVON: Signon

Request signon to the shared variable processor. No other shared variable requests are accepted until **CSVON** has completed successfully.

The fields that must be set are:

PCVREQ **CSVON** (1)

PCVID Processor identification. This is theoretically any 8-byte token, but in practice a fullword integer is normally used, since the APL language supports only such integers when sharing with auxiliary processors. The processor number is expressed as a binary integer (normally less than 1000 for an auxiliary processor) followed by a fullword zero.

PCVECB Pointer to an event control block. The ECB identified will subsequently be posted by the shared variable processor in the following situations:

1. A shared variable offer is being made to this processor.
2. Resources are now available for a **CSVSHARE** which was rejected earlier.
3. The subsystem is asking the processor to sign off.
4. The SVP attempted to post an ECB associated with a share offer made by this processor, but no ECB was specified for the variable.

PCVSPQ Space quota. This is a maximum number of bytes of Shared Memory which the processor might need at any point in time. Shared memory is used for values of variables which have been specified by one partner but not yet referenced by the other. This may be set to a very large positive value; the SVP will respond with the amount of space available.

PCVSHVQ Shared variable quota. This is a maximum number of shared variables which the processor is prepared to handle concurrently. By specifying the maximum possible for this field (32767), the processor can insure that new offers are limited only by SVP resources.

PCVGLB (in **PCVFLGS**) For IBM use only. This flag *must* be *off* for both global and local auxiliary processors. The SVP services will not behave properly otherwise.

The fields set on return are:

PCVRC The return codes are:

-3=CSVEPPF	Processor Table full
00=CSVOK	Normal return
01=CSVENA	SVP not available
04=CSVEASO	Already signed on as PCVID
05=CSVEUSED	Another processor is signed on as PCVID
15=CSVEARG	Argument error

PCVSPQ The maximum amount of space in shared memory available to the processor. This is an upper limit on the size of shared variables. If the processor attempts to set a shared variable to a larger value than this, it will receive a permanent rejection (return code 10) indicating that the value is too large for shared memory. Note that this space is not guaranteed. Even a shorter value may result in a temporary rejection (return code -2) indicating that shared memory is currently full.

PCVSHVQ The maximum concurrent number of shared variables the processor will be permitted to have. If the processor attempts to share more variables than this, it will receive a permanent rejection (return code 6) indicating that the variable quota was exceeded. Note that this quota is not guaranteed. A temporary rejection (return code -2) may occur before reaching the limit. This indicates that shared memory is currently full.

PCVOFFER (in **PCVFLGS**) Set if one or more incoming offers exist at the time of signon.

CSVOFF: Signoff

Request signoff from the shared variable processor. Subsequent SVP requests associated with that processor number (except **CSVON**) will be rejected. Any current offers (matched or unmatched) from the processor will be retracted implicitly.

The fields that must be set are:

PCVREQ **CSVOFF** (2)

PCVID Processor identification. This must match the value used during CSVON.

The only field set on return is:

PCVRC The return codes are:

00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
15=CSVEARG	Argument error

SVP Share Control

Share requests are related to shared variables or share offers. The share requests were listed at the beginning of this chapter and are described in detail in the following subsections.

In all of the share requests, the SVP uses a value called the “pershare index” to associate the request with a specific shared variable. When a new variable is being offered (using **CSVSHARE**), the SVP returns an internally generated pershare index to the caller. It also returns a pershare index for each variable reported in response to **CSVSCAN** or **CSVQUERY**. For all other share requests (including **CSVSHARE** when used to match or inquire) the caller must provide a pershare index previously returned by the SVP.

Each of the share services requires an SCV as a parameter block. An assembler language mapping of the SCV is distributed with the product as macro AP2SCV. Figure 18 shows the structure of that block.

SCV

00	SCVREQ request code		SCVRC return code	
04	SCVPART partner identification			
0C	SCVID processor identification			
14	(reserved)			
18	SCVOSN offer sequence number			
1C	SCVPSX pershare index			
20	SCVECB Event Control Block pointer			
24	SCVVLEN length of variable value			
28	SCVVALUE value assigned to the variable			
2C	SCVACV	SCVFLGS1	SCVFLGS2	SCVNLEN
30	SCVNAME Pointer to name of the variable			
34				

Figure 18. SCV: SVP Share Request Block

The following subsections describe the usage of fields within this parameter block.

CSVCOPY: Copy

Copy the value of a shared variable. Unlike CSVREF, this action does not affect the access state of the variable, nor is it limited by access state.

The fields that must be set are:

SCVREQ **CSVCOPY** (10)

SCVID Processor ID.

SCVPSX Pershare index. Must contain the token returned by **CSVSHARE**.

SCVVLEN Buffer length.

SCVVALUE Pointer to a buffer. The current value of the variable will be returned in this area.

SCVHOLD (in **SCVFLGS1**) Do not release. If set on, the variable is not released on successful completion or on no value found. The variable will remain under the control of the requestor until one of the following requests associated with the same variable completes successfully:

Copy without **SCVHOLD**
Reference - Obtain the value
Release - Release control
Specify - Replace the value
Retract - Retract the variable
Sign off

During the time that a processor retains control of a variable, its share partner will receive return code -1 from any Copy, Reference, or Specify request.

The fields set on return are:

SCVRC The return codes are:

-1=CSVELOCK	Variable interlocked
00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
09=CSVESOF	Buffer size too small
11=CSVENOV	No new value
15=CSVEARG	Argument error
16=CSVENOT	Value cannot be represented

SCVVLEN Set on successful return or CSVESOF error. This field is set to the actual size of the object.

SCVACV The current access state in the high-order four bits, and the combined ACV setting in the low-order four bits.

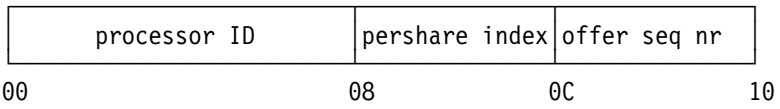
SCVFLGS1 The degree of coupling, indicated by:

SCVFOFR1 (X'01') Offered by this processor.
SCVFSHR (X'02') Fully shared.

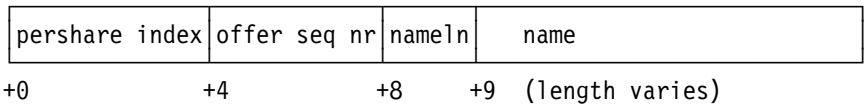
CSVQUERY: Query

Obtain a list of processors or variables whose degree of coupling with the caller matches that specified in the SCV.

If the result is a list of processors, each entry will occupy four fullwords, in the following format:



If the result is a list of variable names, each entry will be byte aligned and variable length in the following format:



where “nameln” is the length of the name which follows it.

The fields that must be set are:

SCVREQ **CSVQUERY** (14)

SCVPART Partner identification. If this field is zero, a list of processors is to be returned. Otherwise, it identifies the processor for which the list of variables is desired.

SCVID Processor identification. Must match the value in **PCVID** during **CSVON**.

SCVOSN Offer Sequence Number. This value limits the list to offers with a sequence number greater than the one specified. Note that on return, the sequence number of the last offer listed will be placed here. Thus if the request is reissued, the list will continue. Specify 0 to begin with the oldest offer.

SCVVLEN Buffer length. The size of the buffer into which the result will be placed.

SCVVALUE Buffer address. Points to the area where names or processor IDs will be returned.

SCVFOFR1 (in **SCVFLGS1**) Offers made by requestor. On to include unmatched outgoing offers.

SCVFOFR2 (in **SCVFLGS1**) Offers made to requestor. On to include unmatched incoming offers.

SCVFSHR (in **SCVFLGS1**) Fully Shared variables. On to include matched shares.

Note: All other flags in **SCVFLGS1** must be off.

The fields set on return are:

SCVRC The return codes are:

-2=CSVESMF	Shared memory full
00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
09=CSVESBS	Buffer size too small
15=CSVEARG	Argument error

SCVOSN The highest offer sequence number of offers reflected in the returned list. Unaltered if no acceptable offers are found.

SCVVLEN The length of the output buffer actually used. This will be zero if no acceptable offers were found. If more acceptable offers are found than will fit in the buffer, a partial result will be returned, and the return code will be **CSVESBS**.

CSVREF: Reference

Reference a shared variable. If no error or interlock occurs, the access state of the variable will be changed to `0 0 1 1`. The current value of the variable will (normally) be moved to the location specified in the SCV.

The fields that must be set are:

SCVREQ **CSVREF** (7)

SCVID Processor ID.

SCVPSX Pershare index. Must contain the token returned by **CSVSHARE**.

SCVVLEN Buffer length. If the length is negative, the access state will be changed and the data will be discarded.

SCVVALUE Pointer to a buffer. The current value of the variable will be returned in this area.

The fields set on return are:

SCVRC The return codes are:

-1=CSVELOCK	Variable interlocked
00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
09=CSVESBS	Buffer size too small
11=CSVENOV	No new value
15=CSVEARG	Argument error
16=CSVENOT	Value cannot be represented

SCVVLEN Set on successful return or **CSVESBS** error. This field is set to the actual size of the object.

SCVACV The current access state in the high-order four bits, and the combined ACV setting in the low-order four bits.

SCVFLGS1 The degree of coupling, indicated by:

SCVFOFR1 (X'01') Offered by this processor.
SCVFSHR (X'02') Fully shared.

CSVREL: Release

Release control of a shared variable. This request may be used following a **CSVCOPY** with the **SCVHOLD** option.

The fields that must be set are:

SCVREQ **CSVREL** (11)

SCVID Processor ID.

SCVPSX Pershare index. Must contain the token returned by **CSVSHARE**.

The fields set on return are:

SCVRC The return codes are:

00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
15=CSVEARG	Argument error

SCVACV The current access state in the high-order four bits, and the combined ACV setting in the low-order four bits.

SCVFLGS1 The degree of coupling, indicated by:

SCVFOFR1 (X'01') Offered by this processor.
SCVFSHR (X'02') Fully shared.

CSVRET: Retract

Retract a shared variable.

The fields that must be set are:

SCVREQ **CSVRET** (12)

SCVID Processor ID.

SCVPSX Pershare index. Must contain the token returned by **CSVSHARE**.

The fields set on return are:

SCVRC The return codes are:

00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
15=CSVEARG	Argument error

SCVFLGS1 SCVFOFR2 (X'04') is set if the variable is still offered to this processor.

CSVSCAN: Scan for an Offer

Search for an outstanding offer to the caller. The first acceptable offer found is used to complete the SCV. Note that the minimum offer sequence number is one of the parameters to this request, so that the request may be repeated to get subsequent offers.

The fields that must be set are:

SCVREQ **CSVSCAN** (3)

SCVPART Partner identification. If zero, all incoming offers will be checked. If nonzero, the scan will be limited to offers from the specified partner.

SCVID Processor identification. Must match the value in **PCVID** during **CSVON**.

SCVOSN Offer sequence number. This value limits the scan to offers with a sequence number greater than the one specified. Note that on return, the sequence number of the offer being described will be placed here. Thus if the request is reissued, the scan will continue. Specify 0 to begin with the oldest offer.

SCVNLEN Name length. The SVP supports names up to 255 characters long. The usage of this field depends on the setting of the **SCVNAMES** flag:

- If **SCVNAMES** is on, this is the length of the *field* pointed to by **SCVNAME**. Note in this case that the value is modified on return, and must be reset before repeating the scan.
- If SCVNAMES is off, this is the length of the *name* pointed to by **SCVNAME**, and only offers with that specific name are scanned.

SCVNAME Pointer to Shared Variable Name field. See **SCVNAMES** for its usage.

SCVNAMES (in **SCVFLGS1**) On if any name is acceptable. Off if the name is specified in the field pointed to by **SCVNAME**.

The fields set on return are:

SCVRC The return codes are:

00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
12=CSVENOF	No Offer found
15=CSVEARG	Argument error

SCVPART The ID of the processor making the offer.

SCVOSN The sequence number of the offer.

SCVPSX The pershare index. This is a token which uniquely identifies the offer, and is used for subsequent requests.

SCVNLEN The length of the name returned.

SCVNAME The pointer is not modified, but if **SCVNAMES** was on in the request, the field it points to will contain the name of the shared variable.

SCVFLGS2 The partner protocol (**1=VSAPL, 2=APL2**). The APL2 interpreter uses protocol **2**, as do as all processors written to the interfaces defined in this manual. Protocol **1** is used by processors originally written to run under the VS APL product, but later moved to the APL2 environment.

CSVSEEAC: See (inspect) Access Information

This service provides quick access to the key attributes of a shared variable, including its access state, access control vector, and degree of coupling.

The fields that must be set are:

SCVREQ **CSVSEEAC** (5)

SCVID Processor ID.

SCVPSX Pershare index. Must contain the token returned by **CSVSHARE**.

The fields set on return are:

SCVRC The return codes are:

00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
15=CSVEARG	Argument error

SCVACV The current access state in the high-order four bits, and the combined ACV setting in the low-order four bits.

SCVFLGS1 The degree of coupling, indicated by:

SCVFOFR1 (X'01') Offered by this processor.
SCVFSHR (X'02') Fully shared.

CSVSETAC: Set ACV

Set the access control vector for a shared variable.

The fields that must be set are:

SCVREQ **CSVSETAC** (6)

SCVID Processor ID.

SCVPSX Pershare index. Must contain the token returned by **CSVSHARE**.

SCVACV Desired access control vector in the low-order four bits. See “Access Control” on page 7.

The fields set on return are:

SCVRC The return codes are:

00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
15=CSVEARG	Argument error

SCVACV The current access state in the high-order four bits, and the combined ACV setting in the low-order four bits. See “Access Control” on page 7.

SCVFLGS1 The degree of coupling, indicated by:

SCVFOFR1 (X'01') Offered by this processor.
SCVFSHR (X'02') Fully shared.

CSVSHARE: Offer a Variable

Note: See also CSVSHARE: Match an Offer and CSVSHARE: Query a Share

This form of the **CSVSHARE** request is distinguished by having **SCVPSX=0**.

The fields that must be set are:

SCVREQ **CSVSHARE** (4)

SCVPART Partner identification. Identification of the processor with which the variable is to be shared. This is normally a fullword integer followed by a fullword zero. If both words are zero, then this offer may be matched by any partner which uses the same name and identifies your processor.

SCVID Processor identification. This must match the **PCVID** field as used during **CSVON**.

SCVOSN Offer sequence number. Only offers with OSN greater than **SCVOSN** can be used to match this offer. If **SCVOSN=0** the effect is to ignore OSN in this call.

SCVPSX Pershare index. Zero for this form of the request.

SCVECB Pointer to an event control block. Points to the ECB which will subsequently be posted on certain actions by the share partner. If **SCVECB** is zero, the **PCVECB** will be used instead for signalling.

SCVVLEN Initial value length. Set this field to zero unless the processor is providing an initial value for the shared variable.

SCVVALUE Pointer to an initial value. (Ignored if **SCVVLEN=0**.)

Note: The value will also be ignored if the share partner has already assigned a value to the shared variable.

SCVACV Access control vector component. This 4-bit value (provided in the low-order 4 bits of the field) will be **OR**ed with the partner's component to produce the combined ACV.

SCVNLEN Name length. The length of the name pointed to by **SCVNAME**. The SVP supports names up to 255 characters long.

SCVNAME Pointer to the field containing the name of the shared variable. A name is required for this form of the request.

SCVNAMES (in **SCVFLGS1**) Off for this form of the request.

The fields set on return are:

SCVRC The return codes are:

-2=CSVESMF	Shared memory full
00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
06=CSVEVQO	Variable quota exceeded
07=CSVESQO	Space quota exceeded
08=CSVESOF	Offer to self
10=CSVEVTL	Value too large for shared memory
15=CSVEARG	Argument error

SCVPART The partner's ID. Normally zero, but will be nonzero if the SVP was able to find an earlier corresponding offer by the proposed partner. See **CPVID** under **CSVON** for the format of this field.

SCVOSN The offer sequence number of the variable. This number provides an ordering of all variables processed by the SVP. It is defined for a variable when the first partner offers it, and is retained until one partner has retracted. At that point the variable reverts to an outstanding offer, and is assigned a new OSN.

SCVPSX The pershare index assigned to the share offer. This is an arbitrary token created by the SVP, which must be used for all subsequent communication regarding the shared variable.

SCVACV The current access state in the high-order four bits, and the combined ACV setting in the low-order four bits.

SCVFLGS1 The degree of coupling, indicated by:

SCVFOFR1 (X'01') Offered by this processor.
SCVFSHR (X'02') Fully shared.

SCVFLGS2 The partner protocol (**1=VSAPL, 2=APL2**) if **SCVFSHR** is set. The APL2 interpreter uses protocol **2**, as do as all processors written to the interfaces defined in this manual. Protocol **1** is used by processors originally written to run under the VS APL product, but later moved to the APL2 environment.

CSVSHARE: Match an Offer

Note: See also CSVSHARE: Offer a Variable and CSVSHARE: Query a Share

This form of the **CSVSHARE** request is distinguished by having **SCVPSX** nonzero, but referring to a variable which is offered to the caller, and not yet matched.

The fields that must be set are:

SCVREQ **CSVSHARE** (4)

SCVPART Partner identification. Identification of the processor with which the variable is to be shared. This is normally a fullword integer followed by a fullword zero. If both words are zero, then this offer will match the offer having the correct PSX and OSN, regardless of partner identification.

SCVID Processor identification. This must match the **PCVID** field as used during **CSVON**.

SCVOSN Offer sequence number. The OSN must be supplied, and must match the OSN of the variable offered. If not, a **CSVENOF** error results. Since the PSX for this call is generally obtained from a **SCAN** request which also returns the OSN, the caller needs only to avoid modifying the **SCVOSN** field between the **SCAN** and **SHARE** requests.

SCVPSX Pershare index. Must be supplied for this form of the request. This field identifies the offer which is being matched.

SCVECB Pointer to an event control block. Points to the ECB which will subsequently be posted on certain actions by the share partner. If **SCVECB** is zero, the **PCVECB** will be used instead for signalling.

SCVVLEN Initial value length. Set this field to zero unless the processor is providing an initial value for the shared variable.

SCVVALUE Pointer to an initial Value. (Ignored if **SCVVLEN=0**.)

Note: The value will also be ignored if the share partner has already assigned a value to the shared variable.

SCVACV Access control vector component. This 4-bit value (provided in the low-order 4 bits of the field) will be **OR**ed with the partner's component to produce the combined ACV.

The fields set on return are:

SCVRC The return codes are:

-2=CSVESMF	Shared memory full
00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
06=CSVEVQO	Variable quota exceeded
07=CSVESQO	Space quota exceeded
08=CSVESOF	Offer to self
10=CSVEVTL	Value too large for shared memory
12=CSVENOF	No offer found
15=CSVEARG	Argument error

SCVPART The partner's ID.

SCVACV The current access state in the high-order four bits, and the combined ACV setting in the low-order four bits.

SCVFLGS1 The degree of coupling, indicated by:

SCVFOFR1 (X'01') Offered by this processor.
SCVFSHR (X'02') Fully shared.

SCVFLGS2 The partner protocol (**1=VSAPL, 2=APL2**). The APL2 interpreter uses protocol **2**, as do as all processors written to the interfaces defined in this manual. Protocol **1** is used by processors originally written to run under the VS APL product, but later moved to the APL2 environment.

CSVSHARE: Query a Share

Note: See also CSVSHARE: Offer a Variable and CSVSHARE: Match an Offer

This form of the **CSVSHARE** request is distinguished by having **SCVPSX** nonzero, and referring to a variable which the caller has already offered or matched.

The fields that must be set are:

SCVREQ **CSVSHARE** (4)

SCVPART Partner identification. Identification of the processor with which the variable is shared, or to which it is offered. This is normally a fullword integer followed by a fullword zero. If both words are zero, the SVP will return that information.

SCVID Processor identification. This must match the **PCVID** field as used during **CSVON**.

SCVPSX Pershare index. Must be supplied for this form of the request. This field identifies the offer about which information is desired.

SCVVLEN Initial value length. This field must be set to zero.

The fields set on return are:

SCVRC The return codes are:

00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
12=CSVENOF	No offer found
15=CSVEARG	Argument error

SCVPART The partner's ID.

SCVACV The current access state in the high-order four bits, and the combined ACV setting in the low-order four bits.

SCVFLGS1 The degree of coupling, indicated by:

SCVFOFR1 (X'01') Offered by this processor.
SCVFSHR (X'02') Fully shared.

CSVSPEC: Specify

Specify a shared variable. If no error or interlock occurs, the current value of the variable will be replaced by the value that the SCV points to.

The fields that must be set are:

SCVREQ **CSVSPEC** (8)

SCVID Processor ID.

SCVPSX Pershare index. Must contain the token returned by **CSVSHARE**.

SCVVLEN Length of the value in bytes.

SCVVALUE Pointer to the Value.

SCVFISPC (in **SCVFLGS1**) Ignore Specification Pending. If on, any unreferenced value set by the partner will be ignored and replaced. If off, and a value set by the partner is waiting, the request will fail with a **CSVEVOS** return code.

The fields set on return are:

SCVRC The return codes are:

-2=CSVESMF	Shared memory full
-1=CSVELOCK	Variable interlocked
00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
07=CSVESQU	Space quota exceeded
10=CSVEVTL	Value too large for shared memory
14=CSVEVOS	Partner's value not read
15=CSVEARG	Argument error

SCVACV The current access state in the high-order four bits, and the combined ACV setting in the low-order four bits.

SCVFLGS1 The degree of coupling, indicated by:

SCVFOFR1 (X'01') Offered by this processor.
SCVFSHR (X'02') Fully shared.

CSVSTATE: State

Request the states of shared variables. A list of pershare indexes is provided in the buffer pointed to by **SCVVALUE**. Each entry is two words long, with the pershare index in the first word. Byte 0 of the second fullword of each entry is filled in with the access state and signal state of the corresponding variable. The access state is in the high-order four bits of this byte, and the signal state is in the low-order four bits.

The fields that must be set are:

SCVREQ **CSVSTATE** (13)

SCVID Processor identification. This must match the **PCVID** field as used during **CSVON**.

SCVVLEN Length of pershare index list.

SCVVALUE Pointer to pershare index list. This is a list of fullwords. If it contains entries with a value of zero, those entries will be skipped without an error indication.

The only field set on return is:

SCVRC The return codes are:

00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
15=CSVEARG	Argument error

CSVDFORM: SVP Data Format Control

This request permits processors to receive data in a format different from that normally used for the processor version. Processors written to the interfaces defined in this manual are APL2-version processors. This request would be used to request data in a VS APL format, or to return to the default APL2 format.

An assembler language mapping of the SDF is distributed with the product as macro AP2SDF. Figure 19 shows its overall structure.

SDF

00	SCVREQ request code	SCVRC return code
04	SCVID processor identification	
0C	SCVPSX pershare index	
10	SCVVERS proc version	SCVDFORM data format
14		

Figure 19. SDF: SVP Data Format Request Block

The request identifies the data format to be used for values subsequently returned by reference or copy requests against the specified shared variable. It does not limit the data formats accepted during shared variable specification.

The fields that must be set are:

SCVREQ **CSVDFORM** (15)

SDFID Processor ID, which must match the **PCVID** field used during signon.

SDFPSX The value returned in **SCVPSX**.

SDFVERS Indicates the protocol being used to interface with the SVP, and hence specifies the nature or the content of **SDFPSX**. This must be **2** (APL2) for processors written to the interface defined in this manual.

SDFDFORM Sets the data format to be used for the specified variable. The valid values are **1** (VS APL) or **2** (APL2).

The only field set on return is:

SDFRC The return codes are:

00=CSVOK	Normal return
01=CSVENA	SVP not available
03=CSVENSO	Processor not signed on
15=CSVEARG	Argument error

Chapter 12. Code T__: Terminal Services

The services in this chapter apply only to Auxiliary Processors. Two terminal services are defined, **TA** which allocates the session terminal, and **TZ** which releases it. Actual terminal I/O must be accomplished with non-APL services such as GDDM* or specific operating system interfaces. APL has no way of verifying that Auxiliary Processors bracket their terminal I/O with proper **TA** and **TZ** calls, but if they do not the results may be visually unpredictable, and asynchronous interrupts may not be handled properly.

The service request codes described in this chapter are:

TA Allocate terminal
TZ Release terminal

TA: Allocate the Terminal

This is a *request* for exclusive use of the terminal. The request returns immediately, whether or not the terminal can be given to the requestor at the moment.

On return, the requesting program should wait for a *resource* signal indicating that the request has been granted. The **PW** service can be used for this purpose.

The requesting process will retain control of the terminal until it explicitly relinquishes that control with a **TZ** request. It may receive a *signal* indicating that some other process is requesting control of the terminal.

```
CALL  (15),(=C'TA',action,resource,signal),
      VL,MF=(E,listarea)
```

Set by processor: None.

Returned to processor

action A one-character field, whose value will be supplied by the service when *resource* is posted. One of the following values will be set:

- **D** Data displayed on the screen has been changed since the processor last controlled it, but field definitions are still valid.
- **F** Field definitions have been changed since the processor last controlled the screen.
- **N** No screen changes have occurred since the processor last controlled the screen, or this is not a full screen terminal, or the processor has never previously controlled the terminal.

resource A fullword ECB which will be posted when the requestor is given control of the terminal. APL will have removed any terminal attention exits.

signal A fullword ECB which will be posted if some other process requests control of the terminal while this process is holding it. The process should release control as soon as possible after being signalled.

TZ: Release the Terminal

```
CALL  (15),(=C'TZ',action),
      VL,MF=(E,listarea)
```

Set by processor

action A one character field indicating what changes have been made to the terminal while it was held:

- **D** Data displayed on the screen has been changed, but field definitions have not.
- **F** Field definitions have been changed.
- **N** No screen changes have occurred.

Returned to processor: None.

Note: If the processor has set any terminal attention exits, it must remove them before issuing this call.

Chapter 13. Code V__: Virtual Storage Services

There are no return codes from these services, except that a returned storage address of zero means the requested storage was not available. If invalid parameters are provided, an ABEND will be issued.

The service request codes described in this chapter are:

VF Free global virtual storage
VG Get global virtual storage
VP Get process virtual storage
VQ Free process virtual storage
VV Get variable length process virtual storage
VX Get extended storage

VF: Free Global Storage

```
CALL  (15),(=C'VF',length,address),
      VL,MF=(E,listarea)
```

Set by processor

length A fullword containing the number of bytes of storage to be freed.

address A fullword containing the address of the storage to be freed.

Returned to processor: None.

Note: Normally only **VG** storage is freed through this service, but it may also be used to free storage obtained using operating system services (subpool 1 for MVS).

VG: Get Global Storage

Storage obtained through this service will not be implicitly freed by APL2. It will not be automatically dumped by **ED** service requests.

The storage is always initialized to binary zero. The storage is always obtained within the 24-bit address range, even when extended addressing is available.

```
CALL  (15),(=C'VG',length,address),
      VL,MF=(E,listarea)
```

Set by processor

length A fullword containing the number of bytes of storage needed.

Returned to processor

address A fullword containing the address of the requested storage. If storage is not available, the address will be set to zero.

Note: This storage may be freed using the **VF** service or operating system main storage services (subpool 1 for MVS).

VP: Get Process Storage

Storage obtained through this service will be implicitly freed when the process terminates. It will also be automatically dumped by **ED** service requests.

The storage is always initialized to binary zero. The storage is always obtained within the 24-bit address range, even when extended addressing is available.

```
CALL  (15),(=C'VP',length,address),
      VL,MF=(E,listarea)
```

Set by processor

length A fullword containing the number of bytes of storage needed.

Returned to processor

address A fullword containing the address of the requested storage. If storage is not available, the address will be set to zero.

Note: This storage may be freed using the **VQ** service. It must not be freed using **VF** or non-APL services.

VQ: Free Process Storage

```
CALL  (15),(=C'VQ',length,address),
      VL,MF=(E,listarea)
```

Set by processor

length A fullword containing the number of bytes of storage to be freed.

address A fullword containing the address of the storage to be freed.

Returned to processor: None.

Note: Only **VP**, **VV** or **VX** storage may be freed through this service . Storage blocks cannot be segmented, i.e. an entire **VP**, **VV** or **VX** area must be freed at once.

VV: Get Variable Length Process Storage

This request is identical to **VP** except that a smaller amount of storage will be accepted if the amount requested is not available.

```
CALL  (15),(=C'VV',length,address),
      VL,MF=(E,listarea)
```

Set by processor

length A fullword containing the maximum number of bytes of storage wanted.

Returned to processor

length A fullword containing the number of bytes actually obtained.

address A fullword containing the address of the requested storage. If storage is not available, the address will be set to zero.

Note: This storage may be freed using the **VQ** service. It must not be freed using **VF** or non-APL services.

VX: Get Extended Storage

Storage obtained through this service will be implicitly freed when the process terminates. It will also be automatically dumped by **ED** service requests.

The storage is not initialized to binary zero. The storage will be obtained above the 16 megabyte line if possible.

```
CALL  (15),(=C'VX',length,address),
      VL,MF=(E,listarea)
```

Set by processor

length A fullword containing the number of bytes of storage needed.

Returned to processor

address A fullword containing the address of the requested storage. If storage is not available, the address will be set to zero.

Note: This storage may be freed using the **VQ** service. It must not be freed using **VF** or non-APL services.

Chapter 14. Code X__: External Call Services

The services in this chapter apply only to processor 11 :LINK.FUNCTION Routines. They provide ways to:

- Build a CDR using a pattern (the **XB** service)
- Convert APL data tokens to addresses (**XC** and **XD** services)
- Evaluate an APL expression (the **XE** service)
- Find or form an APL object (the **XF** service)
- Allocate or free storage in the workspace (the **XG** service)

A number of services in this chapter are called with only two parameters, the service code and an ECV. The ECV is a control block described in "The External Control Vector (ECV)" on page 27. Services called with these parameters do not define the term ECV, but do describe usage of fields within it.

Note: In all cases, an external routine must use the same ECV passed to it when it calls these services.

The service request codes described in this chapter are:

XB build a CDR with a pattern
XC convert data tokens to addresses
XD convert data tokens to address/length pairs
XE evaluate an APL expression
XF form or find an APL object
XG allocate or free space in the workspace

XB: Build a CDR Using a Pattern

In certain situations, it is not possible to predict the format of one or more arguments in advance, and therefore not possible to code a specific **:LARG.** or **:RARG.** tag in the routine description. In these cases, it is possible to specify the argument tag as null (for example, **:LARG.**) or to omit it entirely. If it is specified as null, APL will build a CDR representing the argument without any conversion. If it is omitted, APL will not build a CDR representation of the argument at all. In either case, APL tokens for the left and right arguments are passed to the external function in the ECV "left argument token" and "right argument token" fields (**ECVXTLA** and **ECVXTRA**). Using the **XB** service, CDRs representing the arguments can be built using these tokens and argument patterns.

The format of a pattern CDR matches that of the CDR header and descriptor sections. It is just like a CDR without the pointer or data sections. It consists of the CDRFLAGS, CDRDLEN, CDRXRHO, CDRRT, CDRRL, CDRRANK, and CDRRHO fields only. The contents of the CDRFLAGS field must be valid, but they do not influence the type of CDR produced. A pointer from CDR result or value is always produced.

Unlike a CDR, a pattern CDR may have CDRXRHO or one or more elements of CDRRHO specified as X'80000000' or CDRRANK specified as X'8000'. These values indicate that the corresponding fields are unspecified and are not to be used in rank or shape checking. If CDRRANK is so specified for a particular item of the array, CDRRHO fields may not follow it.

Pattern CDRs are similar to the patterns used in **:RARG.**, **:LARG.**, and **:RSLT.** tags to describe the arguments and results of external routines called through processor 11. See *APL2 Programming: System Service Reference* for more information. Pattern CDRs, however, conform to the true CDR header format (as defined by the AP2CDR macro), while argument patterns are an EBCDIC representation of it.

```
CALL  (15),(=C'XB',ecv),
      VL,MF=(E,listarea)
```

Prior to issuing the **XB** call, the following fields must be set in the ECV:

ECVXPRQT Service request token: the argument token to be converted

ECVXPRQP Service request parameter: the address of the argument pattern CDR.

On completion of the **XB** call, the following fields are set in the ECV:

ECVXCET Result event type: actually two halfwords. This is set to `0 0` if the request was successful or to the event type (⎕*ET*) if unsuccessful.

ECVXPRQT Service request token: the APL token representing the argument CDR which was built. This token should be retained if other service requests are subsequently issued since the address of the argument CDR may change across certain service calls.

ECVXPRQP Service request parameter: the address of the argument CDR built as a result of this call.

ECVXRLOC Relocation count: this value is changed if objects in the workspace have been relocated. After relocation occurs, addresses previously obtained may be invalid. They should be refreshed by reissuing the **XC** or **XD** calls.

XC: Convert Data Tokens to Addresses

This service is used to request conversion of a single data token or all the data tokens in a CDR pointer section to addresses.

- All APL tokens have a high-order zero bit and remain valid across calls to the interpreter, but cannot be used directly to access data.
- Data tokens are found in CDR pointer sections and have a high-order zero bit like object tokens. Hence they are *tokens*
- Data tokens, unlike object tokens, are not associated with APL objects. They are associated with the data described in the CDR.
- Addresses have a high-order one bit and can be used directly to access data, but may need to be refreshed after interpreter calls.

CALL (15),(=C'XC',*token,area[,token,area] ...***),**
VL,MF=(E,*listarea***)**

Set by processor

token This is either:

- A fullword containing a data token that was given to the routine in the pointer section of a CDR. The address of the data associated with the token will be returned.

Or it is:

- A variable length field containing a CDR. (The high-order bit of the first CDR word is **CDRID**, which is always on.) In this case the CDR must contain a pointer section (see "Pointer Section" on page 17). Each of the data tokens in the pointer section will be converted to addresses, and will be returned in the order they appear in the CDR.

Returned to processor

area An output area in which absolute addresses and lengths will be returned. The output area must contain one fullword if a single data token is converted, or one fullword for each pointer entry (of any type) in the CDR. Each of the returned addresses will have the high-order bit set to indicate that they are addresses, not data tokens.

XD: Convert Data Tokens to Address/Length Pairs

This service is similar to **XC** except that both an address and a length is returned for each pointer converted.

```
CALL  (15),(=C'XD',token,area[,token,area] ...),
      VL,MF=(E,listarea)
```

Set by processor

token Either a CDR or a data token within the pointer section that was given to the routine earlier. (See "XC: Convert Data Tokens to Addresses" on page 104 for details.)

Returned to processor

area An output area in which absolute addresses and lengths will be returned. The output area must contain two fullwords if a data token is converted, or two fullwords for each pointer entry (of any type) in the CDR. The address will be returned in the first fullword of each pair, and the length in the second fullword. Each of the returned addresses will have the high-order bit set to indicate that they are addresses, not data tokens.

XE: Evaluate an APL Expression

This service is used to request evaluation of an APL expression. Evaluation of the expression takes place in an environment similar to that of $\square EC$.

Expressions to be executed take the form of tokens in the ECV:

ECVXTLA LA	ECVXTLF LF	ECVXTOP OP	ECVXTRF RF	ECVXTRA RA
left argument	left operand	operator	right operand	right argument

You should provide only the items that are pertinent to the kind of operation you are requesting. Here are the possibilities:

Niladic function				RF	
Monadic function				RF	RA
Dyadic function	LA			RF	RA
Monadic operator deriving dyadic	LA	LF	OP		RA
Monadic operator deriving monadic		LF	OP		RA
Dyadic operator deriving dyadic	LA	LF	OP	RF	RA
Dyadic operator deriving monadic		LF	OP	RF	RA

This may sound quite restrictive, but it really is not at all. The functions and operators we are referring to include ***all*** APL2 primitive, system, and defined functions and operators. So, for example, if you have a character string that you want executed as an APL statement, you can pass the entire string (replete with arbitrary APL2 primitives) as the **RA** and use Φ as the **RF**.

Note: ":LINK.FUNCTION Routine Entry and Exit" on page 27 contains important information on the differences between *tokens* and *addresses*. Keep in mind that

- *tokens* have a high-order zero bit and remain valid across calls to the interpreter, but cannot be used directly to access data.
- *addresses* have a high-order one bit and can be used directly to access data, but may need to be refreshed after interpreter calls.

```
CALL  (15),(=C'XE',ecv),
      VL,MF=(E,listarea)
```

Before executing the request the following fields must be set in the ECV:

ECVXPRQT Service request token: must be set to 0

ECVXPRQP Service request parameter: what to do with the explicit APL result. This may be:

- The address of a pattern CDR to be used to convert the result. (This works like the **XB** service request parameter.)
- X'80000000', to request that a default CDR be built.
- 0, to do no result conversion at this time.

ECVXPRQX Must be zero.

ECVXTLA LA: Left argument token of the expression to be executed, 0 if no left argument

ECVXTLF LF: Left operand token of the expression to be executed, 0 if no left function

ECVXTOP OP: Operator token of the expression to be executed, 0 if no operator

ECVXTRF RF: Right operand token of the expression to be executed, 0 if no right function

ECVXTRA RA: Right argument token of the expression to be executed, 0 if no right argument.

The **LA, LF, OP, RF** and **RA** fields in the ECV may be any of the following:

- Tokens that were arguments to the external routine itself (does not apply to **OP**).
- Tokens obtained through other service requests such as **XG** or **XF**.
- The address of a CDR (except for **OP**).
- The "name" of a primitive function or operator (for **LF, OP,** or **RF**), represented as X'300000' in the first 3 bytes of the relevant field with the EBCDIC symbol for the primitive function or operator in the fourth byte. (The EBCDIC symbol code points are defined in Appendix A of *APL2 Programming: Language Reference*.)

To execute APL expressions containing ordinary names (of variables, defined functions or operators) or distinguished names (of system variables or functions), you may use the Execute primitive (⍎) with the name in its character right argument; or you can obtain the APL tokens for the names using the **XF** service request. For example, an external routine which needed the function provided by $\square TF$ might first issue an **XF** request to acquire the token representing $\square TF$, and then use that token in the **RF** field of a subsequent **XE** request.

Note that it is possible using the **XE** request to cause a recursive call to the external routine. For example, the token representing an external function will be found in the **RF** field when the routine is first entered. If that token becomes involved in an expression requested by an **XE** request, recursion occurs. Recursion may also occur from use of Execute, $\square EA$, or $\square EC$, or by using an **XE** request to invoke a defined function. The external routine must be designed to handle recursive calls if such a situation is possible.

On completion of the service the following fields are set in the ECV:

ECVXCET Result event type: set to a nonzero event type (⎕*ET*) if the expression was not successfully executed or if the expression was executed successfully but its explicit result was not converted successfully to a CDR specified in the service request parameter (**ECVXPRQP**). Possible errors include:

`1 2` If the service request token is nonzero.
`1 12` If bad CDR or pattern CDR supplied.
`2` *nn* If an invalid expression is supplied. (One of the five expression fields is missing or inappropriate.)
any As encountered while executing the expression or converting the result.

ECVXPRQT Service request token:

< 0 If the execution of the expression was not attempted.
= 0 If the expression was attempted and either no explicit result was produced (⎕*ET*=`0 0`) or an error occurred (⎕*ET*≠`0 0`).
> 0 If the expression was executed successfully and an explicit result was produced.

- If conversion of the result was requested, this field provides the APL token for the CDR;
- If the explicit result was not successfully converted to a CDR, or if CDR conversion was not requested, this field provides the APL token for the unconverted result.

You will frequently want to use the **XB**, **XC**, or **XD** services to convert the result token to an address.

ECVXPRQP Service request parameter:

< 0 Address of CDR for a successfully converted result.
= 0 If no result was produced or converted.
> 0 When execution was terminated. This field contains the APL token for a ⎕*EC*-style result which may subsequently be converted to a CDR using the **XB** service.

ECVXRLOC Relocation count: this value is changed if objects in the workspace have been relocated. After relocation occurs, addresses previously obtained may be invalid. They should be refreshed by reissuing the **XC** or **XD** calls.

XF: Form or Find an APL Object

This service is used for the following tasks:

- To obtain the APL token for a derived function
- To find the APL token for an APL named object (defined variable, function, or operator; or system variable or function)
- To assign a value to a named user variable or system variable

```
CALL  (15),(=C'XF',ecv),
      VL,MF=(E,listarea)
```

To obtain the APL token for a derived function the following fields must be set in the ECV:

ECVXPRQT Service request token: must be set to 0

ECVXPRQP Service request parameter: must be set to 0

ECVXPRQX Must be zero.

ECVXTLF LF: token of the left operand

ECVXTOP OP: token of the operator

ECVXTRF RF: token of the right operand or 0 if no right operand

The **LF, OP,** and **RF** token fields may specify an APL token, a CDR address, or a primitive symbol as described for the **XE** service. It is not possible, however, to specify constructs involving brackets (for example, `+/[3]` or `⊂[2]`) although these are available using Execute (`⍎`) with the **XE** service.

On completion of this request, the following fields are set in the ECV:

ECVXCET Result event type: set to `0 0` if successful, or to the event type (`⎕ET`) if not successful. Possible values include:

`2 x` If an invalid expression is supplied.
`1 12` If a bad CDR is supplied.

ECVXPRQT Service request token: contains the APL token for the derived function if successful. If the request was not successful, the contents of this field are unpredictable.

ECVXRLOC Relocation count: this value is changed if objects in the workspace have been relocated. After relocation occurs, addresses previously obtained may be invalid. They should be refreshed by reissuing the **XC** or **XD** calls.

To find the APL token given the name of an APL object the following fields must be set in the ECV:

ECVXPRQT Service request token: the APL token for or address of a type “C1” or “C4” CDR which gives the name of the APL object.

ECVXPRQP Service request parameter: must be set to 0

ECVXPRQX Must be zero.

On completion of this request, the following fields are set in the ECV:

ECVXCET Result event type: set to `0 0` if successful, or to the event type (⎕*ET*) if not successful. Possible errors include:

`1 1` *INTERRUPT* while accessing a shared variable.
`2 2` If the name is malformed or inaccessible.
`3 1` If the name was not found.

ECVXPRQT Service request token: set to the APL token for the value or definition of the named APL object if successful, otherwise unpredictable.

ECVXRLOC Relocation count: this value is changed if objects in the workspace have been relocated. After relocation occurs, addresses previously obtained may be invalid. They should be refreshed by reissuing the **XC** or **XD** calls.

To assign a value to a variable or system variable the following fields must be set in the ECV:

ECVXPRQT Service request token: set to the APL token for or address of a type "C1" or "C4" CDR which gives the name of the variable.

ECVXPRQP Service request parameter: set to the APL token for the value; or the APL token for or address of a CDR representing the value; or if set to X'80000000', then this is start of an Indexed Specification and the value returned is the current value which is to be respecified by a subsequent request. For shared variables, the variable is now locked and failure to respecify will cause unpredictable results.

ECVXPRQX Must be zero.

On completion of this request, the following fields are set in the ECV:

ECVXCET Result event type: set to `0 0` or to the event type (⎕*ET*) if unsuccessful. Common Errors:

`2 3` If name is not a variable or undefined.
`1 12` If the name is malformed or inaccessible or the replacement value is invalid.

ECVXPRQT Service request token: set to the APL token for the value of the object if successful, otherwise unpredictable.

Note: This token is intended **only** for later use by the **XB** or **XE** services. You should not attempt to pass it to the **XC** or **XD** services.

ECVXRLOC Relocation count: this value is changed if objects in the workspace have been relocated. After relocation occurs, addresses previously obtained may be invalid. They should be refreshed by reissuing the **XC** or **XD** calls.

XG: Allocate or Free Space in the Workspace

This service is used to request space from within the workspace or to free space previously obtained with an **XG** service. Space allocated may be used for:

- temporary storage, not needed across calls to the external routine, or
- a CDR (which must be built at the very beginning of the area) that will be passed to some service request or used to return a result to APL.
- a CDR data area (which must also be built at the very beginning of the area).

The space is deleted when the external routine returns.

```
CALL  (15),(=C'XG',ecv),
      VL,MF=(E,listarea)
```

To Allocate Space

Set by processor

ECVXPRQT Service request token: must be set to 0

ECVXPRQP Service request parameter: must be set to the number of bytes to be allocated.

Returned to processor

ECVXCET Result event type: actually two halfwords. This is set to `0 0` if successful or to the event type (□ET) if unsuccessful. Errors which are likely to appear here are:

`1 2` If service request token or parameter is not valid.
`1 3` If the workspace is full.

ECVXPRQT Service request token: set to the APL token for the allocated space if the request was successful. This token should be retained if other service requests are subsequently issued, since the address of the allocated space may change across service calls. This token is used in the free space request to identify the allocated storage to be freed.

ECVXPRQP Service request parameter: set to the address of the allocated space if successful.

ECVXRLOC Relocation count: this value is changed if objects in the workspace have been relocated. After relocation occurs, addresses previously obtained may be invalid. They should be refreshed by reissuing the **XC** or **XD** services.

To Free Space

Set by processor

ECVXPRQT Service request token: the APL token provided (in this same field) when the storage was obtained by an earlier **XG** request.

ECVXPRQP Service request parameter: must be set to 0

Returned to processor

ECVXCET Result event type: actually two halfwords. This is set to `0 0` if successful or to the event type (⎕*ET*) if unsuccessful. The most likely error is `1 2`, if the service request token is not valid.

Note: The token is logically freed, in that its use count is decremented. But since a token may be referenced in other places the physical space may not become available immediately.

Part Three: Using VS APL Processors under APL2

Chapter 15. Extensions to Support New Data Types

VS APL user-written auxiliary processors cannot handle the new APL2 data types (for example, nested or mixed arrays, or complex numbers) unless they are modified. If an unmodified auxiliary processor attempts to reference or copy a shared variable containing a data type not supported by VS APL, it will receive return code 12, reason code 68.

Note: This is a new reason code, which was not documented for VS APL.

To allow VS APL auxiliary processors to handle all APL2 data types, the VS APL SVP interface has been extended in APL2 to support new APL2 data types through the ASVDFORM executable macro and the AP2SDF mapping macro which defines the parameter block used by ASVDFORM.

Note: If you need to use the AP2SDF mapping macro, see Appendix C, "Macros Intended for Customer Use" on page 182 for the information you need to get from your system programmer.

ASVDFORM indicates *data format*, and the macro permits auxiliary processors to receive data from the shared variable processor in either VS APL or APL2 format. APL2 data format is defined in Chapter 3, "APL2 Data Representation" on page 13.

label ASVDFORM SDF={*address* | (*reg*)}

label Optional label for the first executable instruction.

address An RX-type pointer to an SDF control block.

reg Name or number of a register containing the address of an SDF control block.

SDF stands for *share data format*, and the AP2SDF macro maps the parameter block used by ASVDFORM. Its format is shown in the next section.

Before issuing ASVDFORM the caller must have filled in:

SDFID the processor id (as used in PCVID)

SDFPSX the PerShare Index (as returned in SCVPSX)

SDFVERS set to SDFV1, to indicate a VS APL protocol caller

SDFDFORM set to SDFV1 (VS APL) or SDFV2 (APL2), to control the format in which shared variable data is given to the processor

Note that the ASVDFORM macro modifies the data format used for an individual variable, and may be reissued at any time for that variable. If, for example, the processor issues ASVPREF and receives return code 12 68 (the data cannot be represented in VS APL format), it could respond by setting SDFDFORM to SDFV2, issuing ASVDFORM, and then repeating the ASVPREF.

At service completion the return code will be available both in register 15 and in SDFRC. The following return codes can occur:

0 Normal return, no error

24 **SVPINACT** SVP not available

32 **SVPERROR** Argument error

Share Data Format (SDF)

This is a parameter block used with the ASVDFORM call.

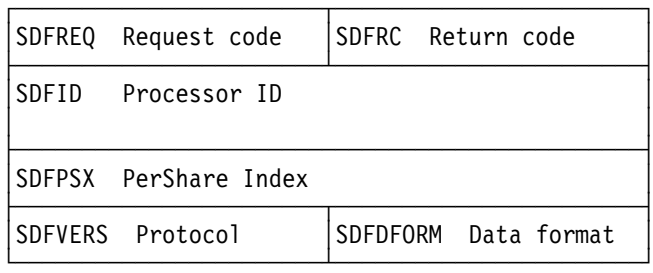
```
AP2SDF [TYPE=DSECT|CSECT],[PRE=prefix]
```

TYPE Optional parameter which can be used to indicate that the parameter block is to be mapped inline without generating a DSECT. The default is TYPE=DSECT.

PRE A 3-character prefix used for all generated labels and the DSECT name. The default is PRE=SDF.

Here is a typical expansion of the SDF:

```
SDF      DSECT
SDFREQ   DS    H        REQUEST CODE
SDFRC    DS    H        RETURN CODE
SDFID    DS    2F       PROCESSOR ID
SDFPSX   DS    F        OFFER SEQUENCE NO.
SDFVERS  DS    H        CALLERS VERSION
SDFDFORM DS    H        DATA FORMAT REQUIRED
SDFLEN   EQU   *-DSF
*
*        VALUES FOR CALLERS VERSION AND DATA FORMAT REQUIRED.
SDFV1    EQU   0001     VSAPL CALLER/DATA FORMAT
SDFV2    EQU   0002     APL2  CALLER/DATA FORMAT
```



Part Four: Calls to APL2 from Non-APL Programs

Chapter 16. Introduction to Calls to APL2

APL2 includes facilities which allow applications written in languages other than APL2 to issue calls to APL2. Such applications can be invoked independently of APL2, or they can be invoked, using facilities provided, from an active APL2 environment.

An interface routine, called *APL2PI* (APL2 Program Interface), provides capabilities through which:

- APL2 can be initialized
- APL2 can be terminated
- APL functions can be executed
- APL variables can be referenced or specified
- APL expressions can be executed
- Control can be passed to an interactive APL2 session.

A companion APL2 external function, called *APL2PIE*, is provided through which:

- Non-APL applications can be invoked from an active APL2 environment. Applications so invoked can subsequently make calls to APL2 using the APL2PI interface.
- A request to terminate can be passed to non-APL applications from the active APL2 environment.
- Control can be returned to a non-APL application that previously invoked APL2 or that returned control to the active APL2 environment.
- Service requests can be passed from executing APL2 functions to any of the currently active non-APL applications.

The APL2PI interface allows applications written in compiled languages to be extended and enhanced with routines written in APL. A wide variety of uses and benefits can be envisaged for such hybrid applications:

- Applications written in languages which do not provide sophisticated numerical computational facilities (for example: COBOL, C) can be enhanced by exploiting APL's power in the area of numerical computation and vector processing;
- Those portions of application which involve complex or changing algorithms might be better or more productively implemented in APL;
- Applications can be prototyped by initially implementing large portions of them in APL, capitalizing on the inherent productivity of APL2 during the application design and implementation phases;
- APL's powerful interactive capabilities can be exploited by applications in which human interaction is an important component. More than just an interactive interface, APL2 offers an interactive computational facility which can be used to substantially enhance compiled applications;
- APL offers distinct benefits for applications which require substantial and frequent changes. Typically, APL2 applications, or those sections of applications written in APL2 can be modified or enhanced much more quickly and at lower

cost than applications or routines written in other languages. By implementing those sections of an application that are most subject to change in APL, the developer can benefit from these characteristics of APL, while retaining the advantages of high level languages for other sections of the application.

The APL2PI routine provides a relatively high level of interface designed to be imbedded as a callable service in programs written in high level languages such as FORTRAN, PL/I, C, or COBOL. APL2PI is a reentrant routine that can be link edited with application programs or packaged as a separate load module which is dynamically loaded before being called. In the VM/CMS environment, APL2PI may also be loaded as a CMS nucleus extension.

Overview of Calls to APL2

The APL2PI interface routine is designed so that it can easily be called from languages such as FORTRAN, COBOL, C, Pascal, PL/I and assembler. The form of such calls (using assembler or FORTRAN syntax) begins with three consistent arguments:

```
CALL APL2PI(request,token,rc,....)
```

where:

request is a 4-character request identifier. The following requests are supported:

INIT	initialize APL2.
TERM	terminate APL2.
APLE	request execution of an APL2 expression.
APLS	request execution of an APL2 function.
APLF	request execution of an APL2 function. This request is more fully functioned than the APLS request, but is not as simple to use.
APLV	reference or specify an APL2 variable.
APLX	return control to the APL2 environment.
APLP	enter or exit the namescope of an APL2 namespace. Subsequent requests will be made in that namescope unless specifically directed elsewhere.

token is a token used by the APL2PI interface for correct and efficient operation. It is returned by an INIT call and should be provided on all subsequent calls.

rc is a 2-element return code returned by the APL2PI interface as the result of any call. A return code of 0 0 indicates success.

Note: The return code is provided as a fullword but often used as a pair of halfwords. See “Return Codes” on page 138.

Most calls to APL2PI require additional arguments specific to the request. These will be described in subsequent sections.

The INIT, TERM, APLE, APLS, APLX, and APLP requests take relatively straightforward arguments that can be easily provided in most high level languages. The APLF and APLV requests, however, are designed to pass arguments to APL2 and

receive results from APL2 in CDR format. CDR format is a data representation which allows efficient representation of APL2 arrays including general arrays. While CDR objects can be constructed in many languages that support data structures (for example: assembler, C, PL/I, Pascal), it is a more difficult format to use than that used in the simpler service requests. The CDR format is described in Chapter 3, “APL2 Data Representation” on page 13.

In the remainder of this section, a simple example will be presented to illustrate the use of this interface. The example will be presented using FORTRAN because of its simple syntax and understandability. The program:

1. defines the necessary data items,
2. causes APL2 to be initialized by means of an INIT service request to APL2PI,
3. prompts the user to enter a set of 3 numbers,
4. computes their average by calling the APL2 function *AVG* in namespace *STATS*,
5. displays the result returned by the APL2 function,
6. causes APL2 to be shutdown by means of a TERM service request to APL2PI.

This example provides overly simplistic error handling facilities (at statement 99), that may not be desirable in an operational environment. More complete examples are shown later.

```
      INTEGER*4 TOKEN,RC,LENGTH
      REAL*8 NUMBERS(3),RESULT
      TOKEN=0
      LENGTH=0
      CALL APL2PI('INIT',TOKEN,RC,'SAMPLE ',0,0,0)
      IF (RC .NE. 0) GOTO 99
      WRITE (6,*) 'Enter 3 numbers'
      READ (5,*) (NUMBERS(I),I=1,3)
      CALL APL2PI('APLS',TOKEN,RC,'STATS ','AVG ',LENGTH,' ',NUMBERS,RESULT)
      IF (RC .NE. 0) GOTO 99
      WRITE (6,*) 'The average is: ',RESULT
      CALL APL2PI('TERM',TOKEN,RC)
      IF (RC .NE. 0) GOTO 99
      RETURN
   99 WRITE (6,*) 'Unexpected error ',RC,' was returned from APL2PI'
      END
```

Figure 20. Sample FORTRAN Program

The *AVG* function invoked by this sample program differs from what a APL2 user might expect:

```
      ∇AVG ARGS;NUMBERS;RESULT
[1]   →(0∨.=3 11 ⎕NA 2 3ρ'PTAATP')/ERROR
[2]   NUMBERS←'E8 1 3' PTA ↑ARGS
[3]   RESULT←(+/NUMBERS)÷ρNUMBERS
[4]   'E8 1 1' ATP RESULT (1↓ARGS)
[5]   →0
[6] ERROR:'UNEXPECTED ERROR' ⎕ES 9 9
    ∇
```

Figure 21. Sample APL2 Function

Lines 2 and 4 of this function use the APL2 external functions *PTA* and *ATP* to retrieve the argument *NUMBERS* passed by the FORTRAN program and to return *RESULT* to that program. These functions will be described in the section entitled "External Functions ATP and PTA" on page 139. Their use is required to accommodate the argument passing mechanisms and data types used in non-APL programs.

Chapter 17. APL2PI Interface Calls

All calls to the APL2PI interface assume that the caller provides the necessary arguments “by reference” using standard OS linkage conventions. That is to say, it is assumed that the calling program passes control to APL2PI with the following general purpose registers set:

R1 contains the address of a standard OS parameter list, that is, a list of the addresses of the arguments passed on the call. The list is terminated by setting the high order bit in the last address in the list. Unless otherwise indicated, all parameters are required. Parameters may only be omitted from the end of the parameter list.

R13 contains the address of a standard 18-word OS save area which will be used by the APL2PI interface.

R14 contains the return address in the calling program.

R15 contains the address of APL2PI.

Assembler (using the CALL macro), FORTRAN, COBOL, and PL/I use these conventions as the default on most calls. C and Pascal, however, often use an extension to these conventions in which a mixture of addresses and values may appear in the parameter list. From C programs, users must ensure that pointers to the arguments, rather than the values of the arguments, are passed. From Pascal programs, users should declare the arguments so that they will be passed by reference rather than by value. Additional information on this subject can normally be found in the programmer's guide manual for the language being used. Some additional information will be provided in later language specific sections of this document.

Many of the arguments required by APL2PI must be specified as character strings, sometimes terminated with a blank. Users should note that many languages, such as PL/I and Pascal, allow definition of variable length character strings which are prefixed with a length field. Such arguments are unacceptable to APL2PI because of the length prefix. Such languages typically provide alternate representations, such as fixed length strings, without the length prefix, that are acceptable to APL2PI. C null terminated character strings are acceptable to APL2PI. If an APL2PI argument must be terminated with a blank, a C null terminated string is acceptable if the character preceding the null is a blank. Again, additional information on this subject can typically be found in the language's programmer's guide manual.

Certain APL2PI arguments (such as *service* on the `'INIT'` call and *pattern* on the `'APLV'` call) are fullword fields containing addresses. Note that in these situations, the caller's parameter list must contain an address that points to the fullword containing the necessary address. All such arguments and result fields will be identified in the following descriptions with a phrase like “...a fullword field containing the address of....”

Each APL2PI call provides as its third argument a fullword field into which APL2PI will place a return code on completion of the call. All such return codes should be interpreted as a pair of halfwords. 0 0 indicates success; 0 x indicates an error originating in APL2PI, or an alternate successful result; x x indicates an error detected by APL2 (rather than in the APL2PI interface) and can be interpreted as

an APL2 □*ET* value. A return code of 1 2 indicates an unexpected *SYSTEM ERROR* that may have been detected by either APL2PI or APL2.

INIT—Initialization Call

```
CALL APL2PI ('INIT',token,rc,name,type,anchor,service,length,parms)
```

This call provides an explicit mechanism by which APL2 can be invoked. If this call is not issued explicitly by the calling program and if APL2 is not active, it will be issued implicitly by other calls to APL2PI (except `'TERM'`). Since invocation of APL2 is often a lengthy process, the calling program might want to issue this call explicitly some time before making use of other APL2PI services.

This call also provides the mechanism by which a non-APL application identifies itself to the APL2PI interface and optionally specifies service routine and anchor addresses. Thus it is recommended that this call be issued by all non-APL applications whether or not APL2 was previously activated.

The arguments to this call are:

`'INIT'` a required argument identifying this request.

token a fullword integer field into which the interface routine will place a token on successful completion of this call. This token should be retained and used on subsequent calls to provide optimal performance. This field should be zero when the `'INIT'` call is issued, or the call will end with an error.

rc a fullword integer field into which the interface routine will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described in “Return Codes” on page 138.

name a name used to identify the calling application program to APL2PI. This name may be subsequently used by the APL2PIE external function to direct requests to this application program. *name* must be 1 to 8 characters in length, and must be terminated with a blank. If this argument is specified as a null or blank, the name `' '` will be assigned to the calling application program. This poses no problem if only one non-APL application uses the APL2PI interface, but may result in errors or unexpected results if more than one non-APL application is activated. A non-blank name is recommended.

type a fullword integer identifying the type of service routine indicated by the *service* argument. A value of 0 means that no service routine is provided; the *anchor* and *service* parameters will be ignored. A value of 1 indicates that the service routine expects its argument and produces its result in non-CDR form. A value of 2 indicates that the service routine expects its argument and produces its result in CDR form. Additional details on service routine arguments and results are presented in section “External Functions APL2PI and APL2PIE” on page 141.

anchor a fullword token passed from the non-APL application. This token will be returned to the non-APL application on every service routine call. Note that updates to this token made during a call to

the service routine will not be retained—the original value of this token will be passed on all service routine calls.

service a fullword containing the address of a routine in the calling application to which service requests can be directed with an *APL2PIE* 3 call from the APL2 environment. If this argument is omitted or specified as 0, or if *type* is specified as 0, *APL2PIE* 3 requests will be denied for this application. See "External Functions APL2PI and APL2PIE" on page 141 for a description of *APL2PIE* 3.

length a fullword integer field specifying the length in bytes of the *parms* argument. If this arguments is omitted or specified with a value of 0, the *parms* argument is ignored.

parms a character string specifying APL2 invocation parameters. This argument is optional, but must be provided if the *length* argument is specified as nonzero.

When an 'INIT' call is issued, if APL2 is not already active, APL2PI will append any invocation parameters provided on the call to the APL2 invocation command provided in AP2XAPIC CSECT (if AP2XAPIC is link edited with APL2PI) or to the default APL2 invocation command:

```
APL2 QUIET RUN(APL2PI)
```

If the resulting invocation options cause an APL2 function other than *APL2PI* to be invoked, that function is expected to invoke the *APL2PI* external function to cause control to be returned to the APL2PI interface routine on completion of APL2 initialization.

If the 'INIT' call is issued when APL2 is already active (i.e.: from a non-APL application invoked via the *APL2PIE* external function), you receive the return code 0 1 (APL already initialized).

TERM—Termination Call

```
CALL APL2PI ('TERM',token,rc)
```

This call requests termination of APL2. It is effective only when issued by the non-APL application from which APL2 was invoked. If issued from a non-APL application which did not cause APL2 invocation (i.e.: one which was invoked by APL2 using the *APL2PIE* external function), it is nilpotent and returns a return code of 0 10 (invalid request).

If APL2 was invoked by a non-APL application, that application must issue the TERM call before its own termination. Failure to do so may cause abnormal termination of APL2, the APL2PI interface and possibly the non-APL application (and possibly even CMS in a VM/CMS environment).

The arguments to this call are:

`'TERM'` a required argument indicating that APL2 is to be terminated.

token a fullword integer containing the token returned on the `'INIT'` call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute. On completion of the `'TERM'` call, this field will be set to zero.

rc a fullword integer field into which the interface routine will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described in “Return Codes” on page 138.

APLS—Execute an APL2 Function

```
CALL APL2PI ('APLS',token,rc,nmspace,fn,rlength,result,arg1,arg2,...)
```

This is one of two calls provided to request execution of an APL function. This call is designed to be easily used in high level language programs (such as FORTRAN or COBOL).

The function specified may reside in a namespace and is called monadically if arguments (*arg1*,*arg2*,...) are specified, or niladically if they are not. Arguments, if any, are passed to the function as a vector of fullword integers which represent the addresses of the argument data. The APL2 function is expected to use *PTA* to access the argument data, and *ATP* to update it. *PTA* and *ATP* are APL2 external functions provided with APL2. They are described in detail in the section entitled “External Functions ATP and PTA” on page 139.

The arguments to this call are:

`'APLS'` a required argument indicating that an APL2 function is to be called.

token a fullword integer containing the token returned on the `'INIT'` call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

rc a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described in “Return Codes” on page 138.

nmspace the name of the namespace in which the specified function is to be found and optionally a surrogate name for the function. If this argument is non-blank,

```
'nmspace' 11 ⎕NA 'fn'
```

or:

```
'nmspace' 11 ⎕NA 'fn SURROGATE'
```

will be executed before the specified function is called. If this argument is coded with an initial blank, no *⎕NA* will be issued before calling the function. Thus, if the function exists in a namespace, the first call to it must provide the *nmspace* argument, but subsequent calls do not. The *nmspace* argument is a character string which is expected to be terminated with a blank, e.g.: `'MYLIB.MYWS '`. If the surrogate name is specified, it must be prefixed with a colon: `'MYLIB.MYWS:SURROGATE '`.

fn the name of the function to be called. This argument is a character string which is expected to be terminated with a blank. It is used as the right argument to *⎕NA* if the *nmspace* argument is non-blank and is then used as the name of the function to be called.

rlength a fullword integer field specifying the length of the result field. On completion of this call, this field will be updated to contain the actual length of the result or error message produced (which may

be shorter, the same size, or longer than the result field). If no explicit result or error message is produced, this field will be set to -1.

Note that this field is normally updated as a result of this call. It therefore should not be coded as a constant on calls from high level languages. This could result in the constant being modified, which in turn could result in subsequent errors in the calling program.

result a field into which the explicit result of the function (if any) will be placed. If the result produced is larger than the length of this field (as specified by *rlength*), only the first *rlength* bytes of the result will be placed into the *result* field and *rlength* will be updated to reflect the actual total result length. The result is placed in this field as an unmodified byte string in left list order, i.e., as if it had been produced by the expression:

```
RESULT←(PFA RESULT) ATR RESULT
```

If an error results from the execution of the specified expression, the error message (⍞←⎕EM) will be placed in the *result* field and its length in the *rlength* field. An error message is not produced in all situations. In general, a message will not be produced if the error is detected before execution of the specified function has begun. In such situations, the *result* field will not be updated.

arg1,*arg2*,... the arguments to the function. The specified function will be passed a vector of integers representing the addresses of these arguments and is expected to use `PTA` to access them and `ATP` to update them. If no arguments are coded in the call, the specified function will be called niladically. A maximum of 64 arguments are supported.

This call allows an explicit result produced by the APL function to be passed back to the calling routine. The calling routine, however, must anticipate the size of this field in advance, allocate storage for it and pass it to APL2PI as the *result* argument. The calling routine cannot control the type, structure, or shape of the data returned, nor can it control whether an explicit result or error is returned. In many situations it may be simpler to pass output or input/output arguments to the APL2 function and structure that function to return its results by updating one or more of the arguments using the APL2 external function `ATP`. This approach allows the *result* field to be used for the return of error information only.

If it is deemed desirable to produce an explicit result in the APL function called, that function can control the data type of the explicit result returned through the use of the external function *ATR*:

```
LENGTH=8
RESULT=0.0
CALL APL2PI('APLS',TOKEN,RC,'STATS ','AVG ',LENGTH,RESULT,NUMBERS)


     ∇RESULT←AVG ARG;NUMBERS
[1]   →(0∨.=3 11 ⎕NA 2 3⍴'ATRPTA')/ERROR
[2]   NUMBERS←'E8 1 3' PTA ARG
[3]   RESULT←(+/NUMBERS)÷⍴NUMBERS
[4]   RESULT←'E8 1 1' ATR RESULT
[5]   →0
[6]  ERROR:'UNEXPECTED ERROR' ⎕ES 9 9
     ∇
```

Figure 22. Sample APLS Call

APLE—Execute an APL2 Expression

```
CALL APL2PI ('APLE',token,rc,slength,string,rlength,result)
```

This call requests execution of an APL2 expression. The expression to be executed is specified as a character string—effectively the right argument of an ⍎ primitive. The result is returned as a byte string derived from the enlist (∈) of the result of the executed expression. If an error occurred during the execution of the expression, the error message (∈⎕*EM*) will be returned in the result field.

The arguments to this call are:

`'APLE'` a required argument indicating that an APL2 expression is to be executed.

token a fullword integer containing the token returned on the `'INIT'` call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

rc a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described in “Return Codes” on page 138.

slength a fullword integer specifying the length of the string to be executed.

string the character string to be executed.

rlength a fullword integer field specifying the length of the result field. On completion of this call, this field will be updated to contain the actual length of the result or error message produced (which may be shorter, the same size, or longer than the result field). If no result or error message is produced, this field will be set to -1.

Note that this field is normally updated as a result of this call. It therefore should not be coded as a constant on calls from high level languages. This could result in the constant being modified, which in turn could result in subsequent errors in the calling program.

result a field into which the result of the executed expression (if any) will be placed. If the result produced is larger than the length of this field (as specified by *rlength*), only the first *rlength* bytes of the result will be placed into the *result* field. *rlength* will be updated to reflect the actual total result length. The result is placed in this field as an unmodified byte string in left list order, i.e., as if it had been produced by the expression:

RESULT←(*PFA RESULT*) *ATR RESULT*

If an error results from the execution of the specified expression, the error message (∈⎕*EM*) will be placed in the result field and its length in the *rlength* field. An error message is not produced in all situations. In general, a message will not be produced if the error is detected before execution of the specified expression has begun. In such situations, the *result* field will not be updated.

APLX—Return Control to APL2

```
CALL APl2PI ('APLX',token,rc,value,result)
```

This call is used to return control to APL2, either to an interactive APL2 session, or to the APL2 application that invoked or transferred control to the non-APL application.

Control may be subsequently returned to the non-APL application by calling either of the APL2 external functions, *APL2PI* or *APL2PIE*.

The arguments to this call are:

`'APLX'` a required argument indicating that control is to be returned to APL2.

token a fullword integer containing the token returned on the `'INIT'` call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

rc a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described in “Return Codes” on page 138.

value this optional parameter, if provided, must be a fullword field containing the value 0 or the address of a CDR. The CDR value must contain three or more items, and the first two must be numeric zeros.

If a CDR is coded, its value will be returned as the explicit result of the *APL2PIE* or *APL2PI* function used to invoke the non-APL application. If the APL2 session was started by the non-APL application, or if *APL2PIE* or *APL2PI* was invoked manually from the APL2 session or when bringing it up, then the CDR value will be displayed, and the user will be left in an interactive APL session.

If no CDR is provided, a three-item result is provided, which will typically display like this:

```
0 0
     +-----------------------------------+
     + Enter 'APL2PI' to return control +
     +-----------------------------------+
```

The default information framed in the result is the text of APL2 message 82, which is translatable. The default may be replaced as follows:

```
'AP2PAPIW' 11 ⎕NA 'MSG'              ⍝ TSO only
'AP2VNRP5.AP2PAPIW' 11 ⎕NA 'MSG'    ⍝ CMS only
 MSG←text_vector
```

result this optional parameter, if provided, must be a fullword field which will be updated to contain the address of a result CDR when and if control is returned to the non-APL application. If control is returned with a call to the *APL2PI* external function, no result CDR is returned, and this field will be set to 0. If control is returned with a call to the *APL2PIE* external function, the left argument to *APL2PIE* will be returned as the result. If no left argument is provided, the field will be set to 0.

If the *result* parameter is not provided, no result will be returned, even if one is provided in the left argument to *APL2PIE*. If *result* is provided *value* must also be provided.

When the 'APLX' call is issued by the non-APL application, the *result* field may contain a fullword zero, in which case any result CDR will be returned without conversion, or it may contain the address of a pattern CDR (see "Pattern CDRs" on page 139), in which case the pattern CDR will be used to convert any result CDR returned.

Execution of an 'APLX' call is not permitted while a namescope entered with the 'APLP' call is the active namescope, and will be rejected with a 0 10 ('invalid request') return code.

APLF—Execute an APL2 Function

```
CALL APL2PI ('APLF',token,rc,nmspace,fn,rslt,larg,rarg)
```

This is the second of two calls provided to request execution of an APL2 function. This call is designed for use from languages such as C and assembler, and it provides greater access to and control over the arguments and result of the specified function. Unlike the `'APLS'` call, this call passes arguments and expects results in APL2 CDR format.

The specified function may be in a namespace and may have any valid valence. The arguments to this call are:

`'APLF'` a required argument indicating that an APL2 function is to be called.

token a fullword integer containing the token returned on the `'INIT'` call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

rc a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described in “Return Codes” on page 138.

nmspace the name of the namespace in which the specified function is to be found and optionally a surrogate name for the function. If this argument is non-blank,

```
    'nmspace' 11 ⎕NA 'fn'
or
    'nmspace' 11 ⎕NA 'fn SURROGATE'
```

will be executed before the specified function is called. If this argument is coded with an initial blank, no ⎕*NA* will be issued before calling the function. Thus, if the function exists in a namespace, the first call to it must provide the *nmspace* argument, but subsequent calls do not. The *nmspace* argument is a character string which is expected to be terminated with a blank, e.g.: `'MYLIB.MYWS '`. If the surrogate name is specified it must be prefixed with a colon: `'MYLIB.MYWS:SURROGATE '`.

fn the name of the function to be called. This argument is a character string which is expected to be terminated with a blank. It is used as the right argument to ⎕*NA* if the *nmspace* argument is non-blank, and it is then used as the name of the function to be called.

rslt when APL2PI is called, this fullword field may be set to 0 or to the address of a pattern CDR (see “Pattern CDRs” on page 139) to be used to convert the result of the APL function. If 0 is specified on the call, the result will be produced as a CDR without conversion. On completion of the APL2PI call, this field will contain 0, if no explicit result or error message was produced, or the address of a CDR representing the result or error message (∈⎕*EM*) produced by the function. If the function completed with an APL2 error, the error message will be returned as a default CDR, without any ref-

erence to the pattern CDR provided on input. An error message is not produced in all situations. In general, a message will not be produced if the error is detected before execution of the specified function has begun. In such situations, the *rslt* field will be set to 0.

larg a fullword field containing the address of the CDR representing the left argument to the function or containing 0 if no left argument is provided.

rarg a fullword field containing the address of the CDR representing the right argument to the function or containing 0 if no right argument is provided.

APLV—Reference or Specify an APL2 Variable

```
CALL APL2PI ('APLV',token,rc,nmspace,variable,value,pattern)
```

This call may be used to obtain or specify the value of an APL variable. If a namespace is specified, the specified variable must already exist in the namespace—it cannot be created with this call. To create a new variable in a namespace, use the `'APLP'` call to enter the namespace namescope, then this call (with no *nmspace* argument) to create the variable and an `'APLP'` call to exit the namespace namescope.

The arguments to this call are:

`'APLV'` a required argument indicating that an APL2 variable is to be accessed.

token a fullword integer containing the token returned on the `'INIT'` call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

rc a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described in “Return Codes” on page 138.

nmspace the name of the namespace in which the specified variable is to be found and optionally a surrogate name for the variable. If this argument is non-blank,

```
    'nmspace' 11 ⎕NA 'variable'
or
    'nmspace' 11 ⎕NA 'variable SURROGATE'
```

will be executed before the specified variable is accessed. If this argument is coded with an initial blank, no `⎕NA` will be issued before accessing the variable. Thus, if the variable exists in a namespace, the first access to it must provide the *nmspace* argument, but subsequent accesses do not. The *nmspace* argument is a character string which is expected to be terminated with a blank, for example, `'MYLIB.MYWS '`. If the surrogate name is specified it must be prefixed with a colon: `'MYLIB.MYWS:SURROGATE '`.

variable the name of the variable to be accessed. This argument is a character string which is expected to be terminated with a blank. It is used as the right argument to `⎕NA` if the *nmspace* argument is non-blank, and it is then used as the name of the variable to be accessed.

value a fullword field containing the address of the CDR representing the value to be assigned to the variable, or containing 0 if the variable is to be referenced. On completion of the call the address of the CDR representing the value of the variable is placed in this field.

pattern This is an optional argument and may be specified when a variable is referenced (*value*=0 on input). If provided, it is a fullword field which may contain the address of a pattern CDR (see “Pattern

CDRs" on page 139) used to convert the value of the variable. If not provided, or if the field contains zero, default conversion will be used to produce the *value* CDR.

APLP—Enter or Exit a Namespace Namescope

```
CALL APL2PI ('APLP',token,rc,nmspace)
```

This call may be executed to enter or exit a specified namespace namescope. Until a namespace namescope is entered, calls to APL2PI will be executed from the namescope established when APL2 was invoked (typically the active workspace namescope). When a namespace namescope is entered via an `'APLP'` call, subsequent calls to APL2PI will be executed in that namescope until another `'APLP'` call is issued to exit the namescope or enter another.

The arguments to this call are:

`'APLP'` a required argument indicating that an APL2 namespace namescope is to be entered or exited.

token a fullword integer containing the token returned on the `'INIT'` call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

rc a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described in “Return Codes” on page 138.

nmspace if non-blank, this argument identifies the namespace whose namescope is to be entered. If blank, the request is to exit the current namescope.

If non-blank, this argument must be terminated with a blank, and must take one of the following forms:

LIBRARY.*MEMBER* or *MEMBER*

where *LIBRARY* is the ddname (TSO) or file name (CMS) of the load library in which the namespace resides, and *MEMBER* is the member name of the namespace. The same rules apply to locating the namespace as when such information is provided in the left argument to □*NA*.

Note that execution of the `'APLX'` call is not permitted while a namescope entered with an `'APLP'` call is the active namescope.

Note also that the uses of `'APLP'` are designed to be paired: an `'APLP'` call to enter a namespace, followed sometime later by an `'APLP'` call (without the namespace specified) to return to the previous environment. Paired `'APLP'` calls can be nested—in other words, one namespace can be entered from another, but care must be taken in unwinding the nesting. An attempt to issue an `'APLP'` call to exit a namespace that was not paired with a previous `'APLP'` call to enter a namespace will result in a 0 10 error return code (invalid request).

The `'APLP'` call operates by accessing □*EA* in the specified namespace. If, when the namespace was created, a list of accessible objects was specified, and if □*EA* was not included in that list, requests to enter that namespace namescope will fail.

Return Codes

Each of the calls described above returns a return code in the *rc* argument field. These return codes are returned as integer fullwords, but are best interpreted as pairs of halfwords. If the first halfword is nonzero, the return code is a $\square ET$ value that resulted from APL2 execution. In addition to the $\square ET$ values, the following return codes are defined:

0 0 Success

0 1 APL is already initialized. This return code may result from an `'INIT'` call and is the expected return code when the `'INIT'` call is issued from a non-APL application which was invoked from APL2 using $APL2PIE$.

0 2 Unexpected shutdown. APL2 has terminated unexpectedly (perhaps as a result of an $)OFF$ command or as a result of an unsuccessful `'INIT'` call). This return code may result from any call other than `'TERM'`, in situations where a non-APL application is running independently of APL2. If this return code is received by a non-APL application invoked from APL, or during processing in a routine nominated as a service routine on an `'INIT'` call, processing should be terminated in an orderly fashion and control returned to the routine's caller.

0 3 Expected shutdown. This return code can result from any call to APL2PI and indicates a request from APL2 for the non-APL application to terminate. In response to the request, the non-APL application should terminate and return control to its caller.

0 4 Insufficient space. There is insufficient free memory for the correct operation of the APL2PI routine. A larger region or virtual machine should be used to run the application.

0 5 (TSO only) Not executing under the TSO TMP. The program which issues calls to APL2PI must be invoked under TSO or a TSO TMP, (typically `IKJEFT01`).

0 10 Invalid request, or invalid parameter list. Among other things, this return code may result from an `'APLX'` call while a namescope, entered with an `'APLP'` call was active, or from an `'APLP'` request to exit a namespace when no namespace namescope was active.

0 11 Unexpected internal error in the APL2PI routine.

Normally, return codes in which the first halfword is nonzero originate from APL2 and should be interpreted as $\square ET$ values. The following, however, can originate from the APL2PI routine:

1 2 System error. This is an unexpected error in the APL2PI routine.

1 5 No shares. The specified namespace cannot be accessed on a `'APLP'` call. This error may occur because the specified namespace could not be located or loaded, or because it was already the active namescope.

3 1 Value error. The variable named on an `'APLV'` call could not be accessed in the specified namespace.

Chapter 18. Using the Calls to APL2 Facility

Using CDR Results

The 'APLX', 'APLF' and 'APLV' calls which return results, return those results in CDR format. These results are always pointer form CDRs and are built as temporary objects in the APL2 workspace. On the next call to APL2PI, these temporary objects are erased before the call is executed. Thus, CDR results returned by APL2PI may not be used as arguments on subsequent calls, and all processing of such results must be performed before any subsequent call to APL2PI.

Pointer form CDRs returned by APL2PI have addresses in the CDR pointer section. That is to say, the CDRPTR fields in that section contain addresses with the high order bit on, and never zeros or tokens.

Pattern CDRs

Pattern CDRs can be specified on 'APLX', 'APLF' or 'APLV' calls to control the creation of the CDR representing the result of an APL2 function or expression or the value of an APL2 variable. If specified, APL2 will attempt to convert the result or value to the data types specified in the pattern CDR. Further, during this conversion, APL2 will check that the ranks and shapes of the result or value and its items correspond to those specified in the pattern CDR. If the result or value cannot be converted as specified, or if a rank or shape mismatch is detected, an appropriate APL2 error will be generated.

The format of CDRs is discussed in Chapter 3, “APL2 Data Representation” on page 13 The format and use of pattern CDRs is discussed in “XB: Build a CDR Using a Pattern” on page 102.

External Functions ATP and PTA

When an 'APLS' call is issued to execute an APL2 function, the arguments provided in the call are passed to the APL2 function as a vector of addresses - one address for each argument. To the APL2 function this appears as a vector of integer values. The external function *PTA* (Pointers to Array) is provided to allow access to these arguments. *PTA* expects a vector of addresses as its right argument and a pattern (similar to the pattern used with the external function RTA) as its left argument, and it will produce an APL2 array as a result. For example:

```
ARRAY←'(G0 1 3)(I4 0)(E8 1 2)(C1 1 10)' PTA POINTERS
```

will convert a set of three arguments—a scalar fullword integer, a pair of double precision real numbers, and a 10 byte character string, respectively—to an APL2 vector of three items.

The external function *ATP* (Array to Pointer) is provided to allow pointer arguments to be replaced (i.e.: updated) with an APL2 array. The syntax for use of this function is:

```
PATTERN ATP ARRAY POINTERS
```

where *PATTERN* is a pattern (similar to the pattern used with *ATR*) which describes the data in the desired format, *POINTERS* is the address of the data to be updated and *ARRAY* is the source array.

Note: This function does not produce an explicit result. Further, it makes no check to ensure that the result fields are large enough to hold the source values.

The *PTA* function assumes a one-to-one correspondence between the data descriptors in the left argument, the data items in the array specified in the right argument, and the set of data areas specified by the pointers in the right argument. Thus, to update a set of three data areas, three pointers must be provided, an array containing three items must be provided, and the pattern must specify either a three element simple array or a general array containing three simple arrays, namely,

```
'C1 1 3' ATP  'ABC' (P1,P2,P3)
```

or:

```
ARRAY←'ABCD' (4 2⍴⍳8) 1.234
PATTERN←'(G0 1 3)(C1 1 4)(I4 2 4 2)(E8 0)'
PATTERN ATP ARRAY (P1,P2,P3)
```

If it is necessary to update a data area with a non-simple APL2 array (i.e.: put a data structure into a single data area), the non-simple array must be converted to a record using *ATR*, and then *ATP* can be used to move it to the data area, namely,

```
ARRAY←'ABCD' (4 2⍴⍳8) 1.234
PATTERN←'(G0 1 3)(C1 1 4)(I4 2 4 2)(E8 0)'
RECORD←PATTERN ATR ARRAY
'C1 1 *' ATP RECORD POINTER
```

Using PTA and ATP

Assume that the function *AVERAGE* in namespace *COMPUTE* is called with the following three arguments:

1. a vector of double precision numbers,
2. a fullword integer indicating the number of items in the first argument,
3. a double precision real field in which the function is to place its result.

The APL2 function might be coded as shown in Figure 23.

```
     ∇AVERAGE ARGS;V;N;R
[1]  →0⍴3 11 ⎕NA 2 3⍴'PTAATP'    ⍝ Access PTA and ATP
[2]  N←'I4 0' PTA 2⊃ARGS         ⍝ Get N from 2nd argument
[3]  V←('E8 1 ',⍕N) PTA ↑ARGS    ⍝ Get V from 1st argument
[4]  R←(+/V)÷N                   ⍝ Compute the average
[5]  'E8 0' ATP R (3⊃ARGS)       ⍝ Update 3rd argument (result)
     ∇
```

Figure 23. Using PTA and ATP

For additional information on the patterns used in the left arguments of *PTA* and *ATP*, see the description of *RTA* and *ATR* in *APL2 Programming: Using the Supplied Routines*, and the description of argument patterns for processor 11 in *APL2 Programming: System Services Reference*.

External Functions APL2PI and APL2PIE

Two APL2 external functions, *APL2PI* and *APL2PIE*, are provided to facilitate communication from APL2 to non-APL applications. *APL2PI* and *APL2PIE* can be accessed by means of ⎕*NA*, namely,

```
0 11 ⎕NA 'APL2PI'
0 11 ⎕NA 'APL2PIE'
```

Note: The first item of the left argument of ⎕*NA* must be 0 (and not 3) for proper operation of the rest of the APL2PI interface.

APL2PI is a niladic function used to return control to the non-APL application after APL2 initialization or after an `'APLX'` call from the non-APL application. It is equivalent to *APL2PIE* `0 ''` as described below.

APL2PIE is an ambivalent function which serves a number of different purposes:

- return control from the APL2 environment to the currently active non-APL application,
- invoke a non-APL application from the APL2 environment,
- request termination of the currently active non-APL application,
- issue a service request to a non-APL application.

Calls to *APL2PIE* can be imbedded in APL2 applications which run independently or are invoked through APL2PI from non-APL applications. Use of *APL2PIE* takes the following forms:

RESULT APL2PIE `0 ''`

Return control to the currently active non-APL application. This request can be issued immediately after APL2 is invoked from a non-APL application or after a non-APL application has returned control to APL2 with an `'APLX'` call. Attempting to return control in any other situation will result in a 1 0 return code. If issued monadically, no result will be returned to the non-APL application. If issued dyadically, the left argument will be returned to the non-APL application in CDR format if the *result* parameter was provided on the `'APLX'` call.

Note that this *APL2PIE* request (or an *APL2PI* request which is equivalent to *APL2PIE* `0 ''`) causes control to be transferred from the APL2 environment at the point at which the request is made. Thus, if a request of this type is imbedded in an APL2 function, the function is suspended at that point and control is transferred to the non-APL application. Subsequent requests from the non-APL application are executed in the context of this suspended function. In particular, a subsequent `'APLX'` request will return control to the suspended function. Users should avoid imbedding *APL2PIE* `0 ''` calls in APL2 applications unless their use is clearly understood and planned for.

COMMAND APL2PIE `1` *NAME*

Invoke a non-APL application using the specified *COMMAND*. *COMMAND* is a character string containing the name of the module to be invoked, optionally followed by one or more arguments to be passed to the module when it is invoked. In the MVS/TSO environment, the specified module must reside in a load library in the user's normal search order. In the VM/CMS environment, the specified module may be the name of an

existing CMS nucleus extension, or the name of a relocatable load module residing on an accessible minidisk.

The non-APL application is assigned the specified *NAME*. That name must match the NAME in any 'INIT' call issued by the non-APL application and in subsequent *APL2PIE* 3 calls to the non-APL application.

When the specified module terminates, control will be returned to APL2 and APL2PIE will return a result of 0 1 RC where RC is the return code resulting from the module. Control may also be returned to APL2 if the non-APL application issues an 'APLX' call. In this case, the result returned by *APL2PIE* will have the form 0 0 MSG, and is provided by the VALUE parameter of the 'APLX' call or a default message provided in the *APL2PAPIW* namespace.

Note that for successful use of this service, APL2PI must be link edited as a separate module and loaded as a nucleus extension in CMS or placed in the APL2 load library in MVS. Additional information on this subject can be found in "Invoking a Non-APL Application through APL2PIE" on page 148.

APL2PIE 2 ''

Request termination of the currently active non-APL application. This request simply sends return code 0 3 back to the non-APL application in response to its last call to APL2PI. The non-APL application is expected to honor this request and terminate. Note that *NAME* may not be specified in the right argument; only the currently active non-APL application can be terminated.

If the non-APL application terminates as expected, a result of 0 1 RC will be returned from APL2PIE, where RC is the termination return code resulting from the non-APL application.

VALUE APL2PIE 3 *NAME*

Make a service request to the *NAME*'d non-APL application. This request is possible only if the non-APL application specified a SERVICE routine address and a TYPE other than 0 on its 'INIT' call. If these requirements are met, this request will cause a subroutine call to that service routine.

If *APL2PIE* is called dyadically, the left argument is passed to the service routine in CDR or non-CDR format depending upon the specification of the TYPE parameter on the 'INIT' call from the non-APL application. In non-CDR format, the *VALUE* will be passed as a byte string result of the expression:

```
(PFA VALUE) ATR VALUE
```

On entry to the service routine of the named non-APL application,

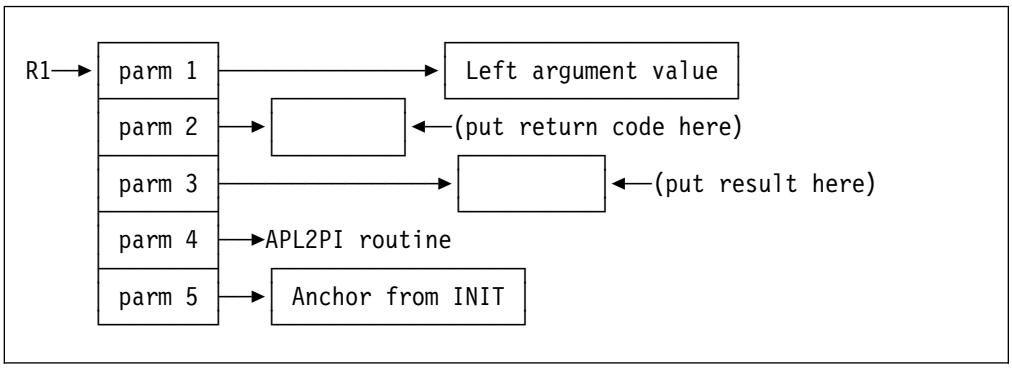


Figure 24. SERVICE Routine Invocation Parameter List

Normal OS calling conventions are used, i.e. register 13 points at an 18-word save area in which the first two words are in use, register 14 contains the return address, and register 15 contains the service routine entry point address.

The return code field is a fullword which can be updated by the service routine. It is initialized to zero. If set to a nonzero value, a three element vector (0,1,return code) will be returned as the result of the *APL2PIE* function call when control is returned from the service routine. The result pointer field is a fullword, initialized to zero, into which the service routine can place the address of a value which will be used as the explicit result of the *APL2PIE* function call when control is returned from the service routine. If the return code is zero, the result pointer field will be examined. If the result pointer is zero, a result of `0 1 0` will be returned. If it is nonzero, the specified result, pointed to by the address in the result pointer field, will be returned.

The result pointer field must contain zero or the address of a result value in CDR or non-CDR form, depending on the specification of the TYPE parameter on the non-APL application's `'INIT'` call. If TYPE was specified as 1 (non-CDR), the result will be interpreted as a byte string prefixed with a fullword length field. If desired, it can be converted to a different form using the APL2 external function *RTA*.

If TYPE was specified as 2 (CDR), the result will be interpreted as an APL2 array in CDR form. The array may have any value, however, if its first item is zero, it must have the form:

N M VALUE

where N M may be 0 0, 0 1, or any defined □*ET* value and *VALUE* may be any valid APL2 array, or may be omitted.

A service request can also result in the following unsuccessful return codes:

0 0 MSG the service routine issued an `'APLX'` request to APL2PI resulting in control being returned to APL2. Such a request may result in unexpected behavior and is not recommended.

0 1 RC the non-APL application terminated with return code RC

1 0 an attempt was made to call the APL2PI or APL2PIE external function when the APL2PI routine was not active or when no non-APL application was active on the APL2PI interface.

0 4	insufficient free space for correct execution
0 10	invalid arguments (typically *NAME* too long)
1 2	unexpected error
1 5	invalid *NAME* or no service routine for the named non-APL application
1 6	this return code results from an attempt to invoke a non-APL application (*COMMAND APL2PIE* 1 '*NAME*') when the named application is already active on the APL2PI interface.

Note that the service routine entered as a result of this call to *APL2PIE*, can issue requests to APL2 using the APL2PI interface. Since such requests can theoretically result in a recursive call to the same service routine, provisions for such an event should be incorporated into the design of the non-APL application.

All of the different calls to APL2PI are supported from executing service routines. It is recommended, however, that the 'TERM' and 'APLX' calls be avoided as they may lead to unexpected and undesired results. The 'TERM' may result in termination of APL2 when the call is made or later at some unexpected time. The 'APLX' call will return control to the APL application that invoked the service routine and it will appear to that application as if the service routine had terminated. If control is subsequently returned to the non-APL application (with an *APL2PI* or *APL2PIE* 0 '' call), control will be returned to the executing service routine which will presumably eventually terminate and return control to APL2.

Note that in many circumstances where the non-APL application and/or the service routine is written in a high level language (such as C or PL/I), *APL2PIE* 3 calls to service routines will not operate correctly. Typically, two problems prevent correct operation:

1. While high level languages provide mechanisms by which a subroutine address can be passed as an argument on a call, the form in which that subroutine address is passed may not be acceptable to APL2PI. The APL2PI INIT call expects the SERVICE parameter to be provided as a fullword which contains the address of the SERVICE subroutine.
2. The linkage conventions expected by a high level language subroutine may not match those provided by APL2PI (as described above) when the service routine is called. Subroutines written in C and PL/I, for example, may require register 12 to be set on entry, and register 13 to point to a save area within a save area stack maintained by the C or PL/I run time environment. These requirements are not fulfilled in the linkage conventions used by APL2PI.

Service routines designed to be used with APL2PI will most typically be written in assembler language. With some limitations, they can be written in FORTRAN (using the IBM* VS FORTRAN program product) if the service routine is structured as a FORTRAN subroutine and the non-APL application is a FORTRAN mainline routine. An example of such a service routine is contained in “Using the APL2PI Interface from FORTRAN” on page 154.

APL2PI and APL2 Calls to Other Languages

Through the use of $\square NA$, APL2 users can invoke applications written in languages other than APL2. Supported languages include C, FORTRAN, PL/I and assembler language, but users have reported success with COBOL and Pascal as well. The APL2PI interface also provides facilities through which APL2 can be invoked by or can invoke a non-APL application.

When using the APL2PI interface, some care must be taken not to interfere with the operation of the non-APL application by calling other non-APL routines through $\square NA$ which are written in the same high level language as the non-APL application interacting with APL2PI.

This situation is of concern because certain high level languages, such as FORTRAN, require access to a “programming environment” for any non-trivial program. Typically, only one instance of the necessary programming environment is supported in a user's address space or virtual machine at any given time. Non-APL applications written in such high level languages that invoke or are invoked by APL2 via the APL2PI interface will typically establish their necessary programming environment as part of their own invocation.

When a non-APL routine is accessed through the use of $\square NA$, the :INIT tag in the routine description for that non-APL routine specifies whether a programming environment is required for correct execution of that routine. If required, Processor 11 will attempt to initialize the environment when the non-APL routine is first called, or as a result of a specific request from the APL2 caller. This instance of the environment is not the same as the instance of the environment established in conjunction with the APL2PI interface, and may not operate correctly or worse, may cause unexpected or erroneous results.

It is therefore recommended that when an application written in a high level language like FORTRAN invokes or is invoked by APL2 via the APL2PI interface, no other routines written in the same language be invoked via APL2PI or $\square NA$. Note that ESSL and OSL routines also have a dependency on the FORTRAN programming environment, and should therefore not be invoked when a non-APL application written in FORTRAN is active on the APL2PI interface.

Additional information on routines accessed through $\square NA$ and their requirements in terms of programming environments can be found in *APL2 Programming: System Services Reference* in the chapter about processor 11 and calling compiled routines.

Chapter 19. System Related Considerations

Using APL2PI in a VM/CMS Environment

In the VM/CMS environment, the APL2PI interface is provided with the following components:

AP2VAPI TXT210
: The object module which contains the APL2PI entry point that is called from non-APL applications. This object module can be combined with the non-APL application or generated as a separate module that can be dynamically loaded by the non-APL application, or accessed as a CMS nucleus extension. Each of these alternatives are described below.

  Note: AP2VAPI TXT210 is the name of the file as it is originally shipped with APL2 Version 2. If maintenance is applied to this file, the filetype will be changed to TEXT. When searching for this file, ensure you have located the most recent copy.

AP2XAPIC AP2MSAMP
: An assembler language source file which can be modified by users to alter the command and parameters used to invoke APL2 from a non-APL application. If an invocation command other than the default:

  ```
  APL2 QUIET RUN(APL2PI)
  ```

  is desired, this source file can be modified, reassembled and combined with the AP2VAPI object module. If AP2XAPIC is combined with AP2VAPI, the invocation command assembled into AP2XAPIC will be used; otherwise, the default invocation command will be used. Note that in either of these cases, the invocation parameters can be supplemented or overridden by means of the PARMS parameter in the 'INIT' call from the non-APL application.

Modifying the APL2 Invocation Command and Options

To change the command or options used to invoke APL2 from a non-APL application:

1. Copy the AP2XAPIC AP2MSAMP file to AP2XAPIC ASSEMBLE
2. Edit AP2XAPIC ASSEMBLE

   Modify the statement labeled APL2CMDN to change the name of the APL2 module invoked. For example, to cause APL2/AE to be invoked, change the statement to:

   ```
   APL2CMDN DC    CL9'APL2AE'                COMMAND
   ```

   Modify the statement labeled APL2CMDO to change the invocation options. It is recommended that the QUIET and RUN(APL2PI) options be left unchanged.
3. After making the necessary changes to AP2XAPIC ASSEMBLE, reassemble it using the following CMS commands:

   ```
   GLOBAL MACLIB AP2MAC
   ASSEMBLE AP2XAPIC
   ```

4. Combine the TEXT file resulting from this assembly with the AP2VAPI object module. This can be done without destroying the original object module with the following CMS commands:

```
COPY AP2VAPI TXT210 A AP2VAPI TEXT A
COPY AP2XAPIC TEXT  A AP2VAPI TEXT A (APPEND
```

   The resulting AP2VAPI TEXT file should then be used in place of AP2VAPI TXT210 in the procedures described below.

Accessing APL2PI from a Non-APL Application

AP2VAPI contains the APL2PI entry point that is called from a non-APL application to request services from APL2. It can be made accessible to the non-APL application in a number of ways:

- If the non-APL application is invoked with CMS LOAD and START commands, the AP2VAPI object module can be made available as a TEXT file on an accessible CMS minidisk, and it will be loaded by CMS when the non-APL application is loaded. To make AP2VAPI accessible as a TEXT file, use the following CMS command:

  ```
  COPY AP2VAPI TXT210 A APL2PI TEXT A
  ```

  Note that it is necessary to change its name to APL2PI TEXT since it is referred to by that name in the non-APL application.

- If the non-APL application is generated as a CMS MODULE using the GENMOD command, APL2PI can be simply incorporated in that module when it is built, namely,

  ```
  COPY AP2VAPI TXT210 A APL2PI TEXT A
  LOAD ... non-APL application ...
  GENMOD ... non-APL application ...
  ```

  APL2PI TEXT will be combined with the non-APL application as a result of the LOAD command.

- In a number of situations, it is may be desirable to structure APL2PI as a CMS nucleus extension and cause it to be loaded and accessed dynamically from the non-APL application. This approach has the advantage that APL2PI is placed in CMS protected storage as an entity separate from the non-APL application. To prepare APL2PI to be loaded as a CMS nucleus extension, it can be converted to a module using the commands:

  ```
  COPY AP2VAPI TXT210 A APL2PI TEXT A
  LOAD APL2PI (CLEAR RLDSAVE
  GENMOD APL2PI
  ERASE APL2PI TEXT A
  ```

  The APL2PI module can be subsequently accessed by the non-APL application using the CMS NUCXLOAD and NUCXDROP commands and NUCEXT functions. For additional information, see *CMS Macros and Functions Reference* and *CMS Command Reference*.

Invoking a Non-APL Application through APL2PIE

The *APL2PIE* external function can be used to invoke a non-APL application which can subsequently make use of the APL2PI interface:

```
0 11 ⎕NA 'APL2PIE'
'COMMAND' APL2PIE 1 'NAME'
```

The user must ensure that when such a request is made, that APL2PI is established as a CMS nucleus extension; otherwise, an error message will be issued by the APL2PIE function and the request will be denied. If APL2 was invoked via APL2PI from a non-APL application, APL2PI will already be established as a CMS nucleus extension, and no other action is necessary. If APL2 was not invoked via APL2PI, then the user must take explicit action to cause it to be established as a nucleus extension.

To explicitly cause APL2PI to be established as a CMS nucleus extension, it must be first created as a CMS module. This can be accomplished with the following CMS commands:

```
COPY AP2VAPI TXT210 A APL2PI TEXT A
LOAD APL2PI (CLEAR RLDSAVE
GENMOD APL2PI
ERASE APL2PI TEXT A
```

The resulting APL2PI MODULE can be established as a CMS nucleus extension with the CMS command:

```
NUCXLOAD APL2PI
```

Parameters may be passed to the non-APL application by specifying them in the *'COMMAND'* left argument of *APL2PIE*, namely,

```
0 11 ⎕NA 'APL2PIE'
'COMMAND PARMS' APL2PIE 1 'NAME'
```

The specified non-APL application (*COMMAND*) is entered with register 0 pointing to a CMS extended parameter list and register 1 pointing to a CMS tokenized parameter list, namely,

```
R0 => A(command verb) => C'COMMAND '
      A(parameters)   => C'PARMS'
      A(end of command)
      A(0)

R1 => CL8'COMMAND '
      CL8'PARMS   '
      X'FFFFFFFF'
```

Extended Addressing Considerations

In virtual machines configured to support 31-bit addressing, APL2 normally runs in 31-bit mode. In that mode, the APL2 workspace is placed above the 16 megabyte line. In order to obtain results from APL2, the APL2PI routine must therefore run in 31-bit mode. This should pose no problem if the non-APL application runs in 31-bit mode when it calls APL2PI, when its service routine is called by APL2PI, and when it is accessing data in CDR form returned by APL2PI.

If the non-APL application must run in 24-bit mode when calling or being called by APL2PI, the APL2 workspace must be forced below the line. This can be done by invoking APL2 with the invocation option XA(24).

If APL2PI is generated as a separate module (so that it can be NUCXLOADed as described above), the correct addressing and residency modes must be specified. If the non-APL application operates in 31-bit mode,

```
GENMOD APL2PI (AMODE 31 RMODE 24
```

is suggested. If the non-APL application operates in 24-bit mode,

```
GENMOD APL2PI (AMODE 24 RMODE 24
```

is required.

Using APL2PI in an MVS/TSO Environment

In the MVS/TSO environment, the APL2PI interface is provided with the following components:

AP2TAPI The object module which contains the APL2PI entry point that is called from non-APL applications. This object module can be link edited with the non-APL application or as a separate load module accessed by the non-APL application.

AP2XAPIC An assembler language source file which can be modified by users to alter the command and parameters used to invoke APL2 from a non-APL application. If any invocation command other than the default:

```
APL2 QUIET RUN(APL2PI)
```

is desired, this source file can be modified, reassembled and link edited with the AP2TAPI object module. If AP2XAPIC is link edited with AP2TAPIC, the invocation command assembled into AP2XAPIC will be used; otherwise, the default invocation command will be used. Note that in either of these cases, the invocation parameters can be supplemented or overridden by means of the PARMS parameter in the 'INIT' call from the non-APL application.

Modifying the APL2 Invocation Command and Options

To change the command or options used to invoke APL2 from a non-APL application:

1. Make a copy of the AP2XAPIC source file from the APL2 distribution data set.
2. Edit your copy of the AP2XAPIC source file.

   Modify the statement labeled AP2CMDN to change the name of the APL2 module invoked. For example, to cause APL2/AE to be invoked, change the statement to:

   ```
   APL2CMDN DC    CL9 'APL2AE'         COMMAND
   ```

   Modify the statement labeled APL2CMDO to change the invocation options. It is recommended that the QUIET and RUN(APL2PI) options be left unchanged.
3. After making the necessary changes to AP2XAPIC, reassemble it, specifying APL2.AP2MACS as the macro library.
4. When link editing AP2TAPI in the procedures described below, specify an INCLUDE statement for the object module produced by this assembly.

Steps 1-3 in this procedure can typically be performed in a straightforward fashion using ISPF (options 3, 2, and 4). If you are unfamiliar with the use of ISPF, consult your system administrator for assistance. Step 4 is typically accomplished by executing a batch job such as the one shown in the next section.

Accessing APL2PI from a Non-APL Application

AP2TAPI contains the APL2PI entry point that is called from a non-APL application to request services from APL2. It can be made accessible to the non-APL application by link editing it with that application, or by link editing it as a separate module and dynamically loading it from the non-APL application.

To link edit it with the non-APL application, simply INCLUDE AP2TAPI (and optionally AP2XAPIC) in the link edit of the non-APL application.

The following job can be used to link edit AP2TAPI (and optionally AP2XAPIC) as a separate load module:

```
//LINK JOB (ACCOUNT),PROGAMER,CLASS=A,TIME=(1),
//         NOTIFY=USERID,MSGCLASS=A,MSGLEVEL=(1,1)
//LINK EXEC PGM=IEWL,REGION=512K,
//         PARM='NCAL,RENT,REUS,MAP,LIST,LET,SIZE=(512K,64K)'
//SYSPRINT DD SYSOUT=*
//SYSLMOD  DD DISP=SHR,DSN=output data set
//OBJ      DD DISP=SHR,DSN=input data set
//SYSUT1   DD UNIT=SYSDA,SPACE=(13030,(40,20))
//SYSLIN   DD *
 MODE AMODE(31),RMODE(ANY)
 INCLUDE OBJ(AP2TAPI)
 INCLUDE OBJ(AP2XAPIC)
 ENTRY APL2PI
 NAME APL2PI(R)
/*
//
```

The input data set should identify the dataset in which AP2TAPI and AP2XAPIC reside. The output dataset must be available to the non-APL application when it needs to load APL2PI. The APL2 load library provides a convenient location since that dataset must also be available during execution of the APL2PI interface.

Under MVS, APL2 and the APL2PI routine must be invoked under the TSO terminal monitor program, IKJEFT01. This is the normal mode of operation if the application is running in a TSO environment, and no special action is needed. If, however, the application was designed to operate in a batch environment, it must be invoked through the TSO terminal monitor program. For example, a batch program, normally invoked with:

```
//STEP   EXEC PGM=MYPROG
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
```

must instead be invoked with:

```
//STEP   EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
MYPROG
/*
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
```

Finally, in either a TSO or batch environment, the files necessary to run APL2 must be allocated before issuing the APL2PI 'INIT' call from the non-APL application.

Invoking a Non-APL Application through APL2PIE

The *APL2PIE* external function can be used to invoke a non-APL application which can subsequently make use of the APL2PI interface, namely,

```
0 11 ⎕NA 'APL2PIE'
'COMMAND' APL2PIE 1 'NAME'
```

The user must ensure that when such a request is made, that APL2PI is available as a separate load module in the same load library (or in the same concatenated list of libraries) from which APL2 was loaded. Instructions on creating APL2PI as a separate load module are described above.

Parameters may be passed to the non-APL application by specifying them in the '*COMMAND*' left argument of *APL2PIE*, namely,

```
0 11 ⎕NA 'APL2PIE'
'COMMAND PARMS' APL2PIE 1 'NAME'
```

The specified non-APL application (*COMMAND*) is entered with register 1 pointing to a TSO CPPL control block. The first word in this control block points to the TSO command buffer representing this command, namely,

```
R1 => A(command buffer) => command buffer
```

The command buffer begins with a halfword length field indicating the total length in bytes of the command buffer, followed by a halfword offset field indicating the offset from the command name to the beginning of the command parameters, followed by the command. Thus for '*COMMAND PARMS*' *APL2PIE* 1 '*NAME*',

```
R1 => A(command buffer) => X'00170008',C'COMMAND PARMS'
```

Extended Addressing Considerations

In the XA or ESA environments, APL2 normally runs in 31-bit mode. In that mode, the APL2 workspace is placed above the 16 megabyte line. In order to obtain results from APL2, the APL2 routine must therefore run in 31-bit mode. This should pose no problem if the non-APL application runs in 31-bit mode when it calls APL2PI, when its service routine is called by APL2PI, and when it is accessing data in CDR form returned by APL2PI.

If the non-APL application must run in 24-bit mode when calling or being called by APL2PI, the APL2 workspace must be forced below the line. This can be done by invoking APL2 with the invocation option XA(24).

Chapter 20. Language Related Considerations

This chapter provides specific information and examples about using the calls to APL2 facility from a variety of languages. The section for each language refers to the APL2 function shown in Figure 25.

```
      ∇AVG ARGS;SIZE;NUMBERS;RESULT
[1]   ⍝ ARGS: Vector of 3 addresses from non-APL application
[2]   ⍝  ARGS[1]→ Number of numbers  (fullword integer)
[3]   ⍝  ARGS[2]→ Vector of numbers  (floating point)
[4]   ⍝  ARGS[3]→ Result field (fullword integer)
[5]
[6]    →(0∨.=3 11 ⎕NA 2 3⍴'PTAATP')/ERROR
[7]    →(3≠⍴ARGS)/ERROR
[8]
[9]   ⍝ Retrieve size of input vector from ARGS[1]
[10]   SIZE←'I4 0' PTA ARGS[1]
[11]
[12]  ⍝ Retrieve vector of numbers from ARG[2]
[13]   NUMBERS←('E8 1 ',⍕SIZE)PTA ARGS[2]
[14]
[15]  ⍝ Compute the average
[16]   RESULT←(+/NUMBERS)÷SIZE
[17]
[18]  ⍝ Return result to ARGS[3]
[19]   'E8 0' ATP RESULT ARGS[3]
[20]   →0
[21]
[22] ERROR:'Unexpected error' ⎕ES 9 9
     ∇
```

Figure 25. APL2 AVG Program Used in Language-Related Examples

Using the APL2PI Interface from FORTRAN

Most functions available on the APL2PI interface can be used in a simple and straightforward fashion in FORTRAN programs. Since the FORTRAN language does not provide support for data structures or pointers however, the `'APLF'` and `'APLV'` calls cannot be used effectively, and only limited function is available between APL2 and a FORTRAN service routine.

This section presents three simple examples of the use of the APL2PI interface from FORTRAN programs. Each of the examples shown has the ability to invoke APL2, or be invoked by APL2. The IBM VS FORTRAN Program Product (5668-805) Version 2, Release 4 was used to construct these examples. Other FORTRAN compilers may or may not have similar capabilities.

The first example shows a FORTRAN program which makes use of the APL2PI `'APLS'` call to invoke the APL2 function *AVG* in namespace *PKGLIB.STATS* to obtain the average of a vector of numbers passed to it. Lines of the FORTRAN routines shown below are numbered on the left for reference in the notes after the figure.

```
 1        REAL*8 NUMBERS(1000),RESULT
 2        INTEGER*4 TOKEN,RC,SIZE,LENGTH
 3        INTEGER*2 RETCODE(2)
 4        EQUIVALENCE (RC,RETCODE(1))
 5        TOKEN=0
 6        LENGTH=0

    C     ---- CALL APL2PI TO INITIALIZE APL2
 7        CALL APL2PI('INIT',TOKEN,RC,'SAMPLE ',0,0,0,16,'SM(OFF) WS(200K)')
 8        IF (RC .GT. 1) GOTO 98

 9        WRITE (6,*)'Enter number of numbers to average'
10        READ  (5,*) SIZE
11        WRITE (6,1)'Enter ',SIZE,' numbers'
12        READ  (5,*) (NUMBERS(I),I=1,SIZE)

    C     ---- CALL APL2PI TO COMPUTE AVERAGE
13        CALL APL2PI ('APLS',TOKEN,RC,'PKGLIB.STATS ','AVG ',LENGTH,' ',
         1             SIZE,NUMBERS,RESULT)
14        IF (RC .NE. 0) GOTO 99

15        WRITE (6,2) 'The average is: ',RESULT

    C     ---- CALL APL2PI TO TERMINATE APL2
16     10 CALL APL2PI ('TERM',TOKEN,RC)
17        RETURN

    C     ---- UNEXPECTED ERROR FROM APL2 INITIALIZATION
18     98 WRITE (6,*) 'Unexpected error during APL2 initialization'
20        WRITE (6,3) 'Return code: ',(RETCODE(I),I=1,2)
21        RETURN

    C     ---- UNEXPECTED ERROR DURING APL2 FUNCTION EXECUTION
22     99 WRITE (6,*) 'Unexpected error during APL2 function execution'
23        WRITE (6,3) 'Return code: ',(RETCODE(I),I=1,2)
24        GOTO 10

25      1 FORMAT(' ',A,I2,A)
26      2 FORMAT(' ',A,F8.3)
27      3 FORMAT(' ',A,2I2)

28        END
```

Figure 26. FORTRAN Program Demonstrating Use of the APLS Request

Notes

- Lines 1-2: define the various data items that will be passed as arguments in subsequent APL2PI calls. It is important to ensure that the data types defined match those expected by APL2PI (e.g.: `TOKEN`, `RC`, `L`) and by the APL2 function being called (`SIZE`, `NUMBERS`, `RESULT`).
- Lines 3-4: the return code returned by APL2PI is returned in the fullword (`INTEGER*4`) field `RC`. This code is, however, best interpreted as a pair of halfwords (`INTEGER*2`). The definition of `RETCODE` as `INTEGER*2` and equivalent to `RC` provides simple and meaningful subsequent access to the return code.

- Lines 7-8: APL2 is initialized with an `'INIT'` call to APL2PI. The FORTRAN program is identified to APL2PI as `'SAMPLE'`. Since no service routine is required, the `type`, `anchor`, and `service` parameters are coded as `0` on the call.

  If APL2 is not active when these statements are executed, it will be initialized as a result of this call and a return code of 0 will be returned. If APL2 was already active, this call is used to identify the FORTRAN program to APL2PI and a return code of 1 will be returned. Thus the error routine is only invoked if an unexpected error is encountered.

- Line 13: The function *AVG* in namespace *PKGLIB.STATS* is invoked with arguments `SIZE`, `NUMBERS` and `RESULT`. Since this APL2 function is expected not to return any explicit result, a value of `0` is passed as the `rlength` parameter on the call. Because this field is updated to reflect the actual length of the result, the parameter cannot be coded as `0` in the call itself. Instead, the variable `L` is defined and initialized to `0` prior to the call. If this call was made repetitively, `L` would have to be reinitialized prior to each call. Failure to do so could cause unexpected results in the FORTRAN program.

- Line 16: Note that the return code from the `'TERM'` call is not checked. Two possible return codes might be expected: 0 0 if the non-APL application originally caused APL2 to be invoked, or 0 10 if the non-APL application was invoked from an active APL2 environment.

- Lines 6, 10, 13 and 19 in the APL2 function *AVG*: FORTRAN passes arguments to subroutines “by reference.” That is to say, FORTRAN passes the addresses of argument data rather that the values of the argument data. The external function *PTA* allows an APL2 application to retrieve data passed by reference, and the external function *ATP* allows an APL2 application to update arguments which were passed by reference.

The second example demonstrates the use of the `'APLX'` and `'APLE'` calls. When this program is executed, it causes APL2 to be initialized and then returns control to the APL2 environment. When this occurs, the following message will appear on the user's screen:

```
0 0
      +-----------------------------------+
      + ENTER 'APL2PI' TO RETURN CONTROL  +
      +-----------------------------------+
```

At this point the APL2 user can interact with APL2 freely and, when finished returns control to APL2PI by calling the APL2PI external function:

```
      A←B←C←10
      APL2PI
```

```
 1        CHARACTER RESULT(1000)
 2        INTEGER*4 TOKEN,RC,LENGTH
 3        INTEGER*2 RETCODE(2)
 4        EQUIVALENCE (RC,RETCODE(1))
 5        TOKEN=0
 6        LENGTH=1000

    C     ---- CALL APL2PI TO INITIALIZE APL2
 7        CALL APL2PI('INIT',TOKEN,RC,'SAMPLE ',0,0,0,16,'SM(OFF) WS(200K)')
 8        IF (RC .GT. 1) GOTO 99

    C     ---- RETURN CONTROL TO APL2
 9        CALL APL2PI('APLX',TOKEN,RC)
10        IF (RC .NE. 0) GOTO 98

    C     ---- EXECUTE AN APL2 EXPRESSION
11        CALL APL2PI('APLE',TOKEN,RC,13,''' ''',□NL 2 3 4',LENGTH,RESULT)
12        IF (RC .NE. 0) GOTO 97

13        WRITE (6,*) 'Names in APL2 workspace: ',(RESULT(I),I=1,L)

    C     ---- CALL APL2PI TO TERMINATE APL2
14     10 CALL APL2PI ('TERM',TOKEN,RC)
15        RETURN

    C     ---- UNEXPECTED ERROR FROM APL2 INITIALIZATION
16     99 WRITE (6,*) 'Unexpected error during APL2 initialization'
17        WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
18        RETURN

    C     ---- UNEXPECTED ERROR ON ATTEMPT TO RETURN CONTROL TO APL2
19     98 WRITE (6,*) 'Unexpected error returning control to APL2'
20        WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
21        GOTO 10

    C     ---- UNEXPECTED ERROR ON DURING APL2 EXECUTION
22     97 WRITE (6,*) 'Unexpected error during APL2 execution'
23        WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
24        GOTO 10

25      1 FORMAT(' ',A,2I2)

26      END
```

Figure 27. FORTRAN Program Demonstrating Use of APLX and APLE Requests

Notes

- Line 9: causes control to be returned to the active APL2 session. The `value` and `result` optional parameters on this call cannot be used in a FORTRAN program. This is because these parameters involve the use of data in CDR format, and FORTRAN is not capable of dealing with CDR format.

- Line 11: causes the APL2 expression

  ```
  ' ',⎕NL 2 3 4
  ```

  to be executed. Catenating a blank on to the output of ⎕NL causes the names to be separated by at least one blank in the result returned to the FORTRAN program. Note that APL2 characters may be imbedded in FORTRAN source programs but may not be printed correctly in the compiled listings.

The third example demonstrates the use of a FORTRAN service routine which can be accessed from APL. In this example, the FORTRAN mainline routine initializes APL2, specifying the SERVICE subroutine as a type 1 service routine to be used by the APL2PI interface. After APL2 initialization is complete, the FORTRAN mainline then passes control to the APL2 environment with an 'APLX' call. When that happens, the following message appears on the APL2 user's screen:

```
0 0
      +-----------------------------------+
      + ENTER 'APL2PI' TO RETURN CONTROL  +
      +-----------------------------------+
```

and the user can interact with APL2 freely. For the purposes of this example, the user should enter the following to cause the SERVICE subroutine to be invoked:

```
      0 11 ⎕NA 'APL2PIE'
      1234 APL2PIE 3 'SAMPLE'
```

This causes control to be passed to the service routine (the SERVICE subroutine) of the APL2PI application identified as SAMPLE (the mainline FORTRAN program). The value 1234 is the first of 5 arguments passed to the service routine. All of the arguments can be retrieved by the FORTRAN SERVICE routine, but only the return code argument can be updated to return data from the FORTRAN SERVICE routine to the APL2 environment.

Once execution of the SERVICE routine is complete, and control is passed back to the APL2 user, the user completes his work and returns control to the FORTRAN mainline by calling the APL2PI external function.

Notes

- Line 8: the fourth argument of the 'INIT' call causes the FORTRAN application to be identified to APL2PI with the name SAMPLE. This name will be subsequently used as the second item of the right argument of *APL2PIE* when control is passed to the SERVICE routine, whose address is provided in the seventh argument of the 'INIT' call.

```
 1        EXTERNAL SERVICE
 2        CHARACTER RESULT(1000)
 3        INTEGER*4 TOKEN,RC,LENGTH
 4        INTEGER*2 RETCODE(2)
 5        EQUIVALENCE (RC,RETCODE(1))
 6        TOKEN=0
 7        LENGTH=1000

    C     ---- CALL APL2PI TO INITIALIZE APL2
 8        CALL APL2PI('INIT',TOKEN,RC,'SAMPLE ',1,0,SERVICE,16,'SM(OFF) WS(200K)')
 9        IF (RC .GT. 1) GOTO 99

    C     ---- RETURN CONTROL TO APL2
10        CALL APL2PI('APLX',TOKEN,RC)
11        IF (RC .NE. 0) GOTO 98

    C     ---- EXECUTE AN APL2 EXPRESSION
12        CALL APL2PI('APLE',TOKEN,RC,13,''' ''',□NL 2 3 4',LENGTH,RESULT
13        IF (RC .NE. 0) GOTO 97

14        WRITE (6,*) 'Names in APL2 workspace: ',(RESULT(I),I=1,L)

    C     ---- CALL APL2PI TO TERMINATE APL2
15     10 CALL APL2PI ('TERM',TOKEN,RC)
16        RETURN

    C     ---- UNEXPECTED ERROR FROM APL2 INITIALIZATION
17     99 WRITE (6,*) 'Unexpected error during APL2 initialization'
18        WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
19        RETURN

    C     ---- UNEXPECTED ERROR ON ATTEMPT TO RETURN CONTROL TO APL2
20     98 WRITE (6,*) 'Unexpected error returning control to APL2'
21        WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
22        GOTO 10

    C     ---- UNEXPECTED ERROR ON DURING APL2 EXECUTION
23     97 WRITE (6,*) 'Unexpected error during APL2 execution'
24        WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
25        GOTO 10

26      1 FORMAT(' ',A,2I2)

27        END

28        SUBROUTINE SERVICE(VALUE,RC,RESULT,ADDRESS,ANCHOR)
29        INTEGER*4 VALUE,RC,RESULT,ADDRESS,ANCHOR
30        WRITE (6,*) 'FORTRAN SERVICE routine called by APL2'
31        WRITE (6,*) 'Input values: ',VALUE,RC,RESULT,ADDRESS,ANCHOR
32        RC=9999
33        RETURN
34        END
```

Figure 28. FORTRAN Program Demonstrating Use of a Service Routine

Using the APL2PI Interface from C

Most functions available on the APL2PI interface can be used in C programs. The 'APLF' and 'APLV' calls and some variants of the 'APLX' call involve the use of data in CDR format and are more difficult, but not impossible, to handle in the C environment. C service routines are not supported.

This section presents a number of examples of the use of the APL2PI interface from C programs. Each of these examples has the ability to invoke APL2, or be invoked by APL2. The IBM C/370 Program Product (5688-039, 5688-040) Version 1 Release 2 was used to construct these examples. Other C compilers may or may not have the same capabilities.

To understand any of the calls to APL2PI from the C environment, the reader must understand the linkage conventions used by the APL2PI interface. All calls to or from the APL2PI interface assume the /370 OS linkage convention. That is to say, when the call occurs, it is expected that the caller will have set the following registers:

- R1 contains the address of the caller's parameter list. The parameter list is expected to contain a list of addresses—one for each argument in the call.
- R13 contains the address of a save area, 18 fullwords in length, which may be used by the called routine to save the caller's registers.
- R14 contains the return address in the calling routine.
- R15 contains the entry point address in the called routine.

To cause the C program to utilize these conventions, `#pragma linkage(...,OS)` must be used in the C program to define the APL2PI routine.

Further, any arguments which are to be updated by the called routine must be passed as pointers rather than values. In the C language, arrays and strings are always passed as pointers, so they require no special handling. Scalars arguments, however, are not normally passed as pointers and must be prefixed with & if they are to be updated, namely,

```
#pragma linkage(ROUTINE,OS)
int input,output,array[5]
input=3
ROUTINE(input,&output,array)
```

causes the procedure ROUTINE to be called with arguments input, output and array. The arguments output and array can be updated by the called ROUTINE, but any attempt to update input will not be reflected in the calling program.

Finally, APL2PI expects to be called as a subroutine rather than as a function and thus produces no explicit result.

The first example shown below illustrates a C program which makes use of the APL2PI 'APLS' call to invoke the APL2 function *AVG* (see Figure 25 on page 153) in namespace *PKGLIB.STATS* to obtain the average of a vector of numbers passed to it. Lines of the C program are numbered on the left for reference in the notes after the figure.

```
 1  #pragma linkage(APL2PI,OS)
 2  #include <stdio.h>
 3  main()
 4  {
 5      int token=0,size;
 6      union {
 7         int code;                    /* Return code as 1 fullword  */
 8         short rc[2];       /* Return code as 2 halfwords */
 9      } rc;
10      double numbers[1000],result,value;
11      char parms[]="SM(OFF) WS(200K)"; /* APL initialization parms */
12      int len=sizeof(parms)-1;          /* Length of parms          */

    /*  ---- Call APL2PI to initialize APL2 ----  */
13      APL2PI("INIT",&token,&rc.code,"SAMPLE ",0,0,0,len,parms);
14      if(rc.code > 1) goto error1;

15      printf("Enter numbers to be averaged\n");
16      printf("Terminate input with non-numeric\n");
17      for (size=0 ; 0<scanf("%lf",&value) ;size++)
18          numbers[size]=value;

    /*  ---- Call APL2PI to compute average ---- */
19      APL2PI("APLS",&token,&rc.code,"PKGLIB.STATS ","AVG ",0,' ',
20             size,numbers,&result);
21      if(rc.code != 0) goto error2;

22      printf("\nThe average is: %lf\n",result);

    /*  ---- Call APL2PI to terminate APL2 ----  */
23     shutdown:
24      APL2PI("TERM",&token,&rc.code);
25      return 0;

26     error1:
27      printf("Unexpected error during APL2 initialization\n");
28      printf("Return code: %hd %hd\n",rc.rc[0],rc.rc[1]);
29      return;

30     error2:
31      printf("Unexpected error during APL2 function execution\n");
32      printf("Return code: %hd %hd\n",rc.rc[0],rc.rc[1]);
33      goto shutdown;

34  }
```

Figure 29. C Program Demonstrating Use of the APLS Request

Notes

- Line 1: causes APL2PI to be called with OS linkage conventions.
- Line 6-8: the return code produced by APL2PI is returned as a vector of 2 halfwords. Redefinition of the return code field allows simpler comparison to expected values such as 0 0 or 0 1.

- Lines 13, 19, 24: the scalar arguments `token` `rc` and `result` are updated by APL2PI and so must be prefixed with `&`.

The following example demonstrates the use of the `'APLP'` and `'APLF'` calls in a C program and the use of data in CDR format. This example also shows that the `'APLF'` call can be used to request execution of APL2 primitives or system functions.

```
 1  #pragma linkage(APL2PI,OS)
 2  #include <stdio.h>
 3  #define CDRID 0x80000000
 4  main() {

 6      struct cdrdesc {               /* CDR Descriptor section */
 7          union {
 8              unsigned int  cdrdlen;
 9              unsigned char cdrflags;
10          } cdrhdr;
11          int   cdrxrho;
12          char  cdrrt;
13          char  cdrrl;
14          short cdrrank;
15      };

16      struct cdrptr {                /* CDR Pointer section    */
17          int cdrpslen;
18          int cdrdslen;
19          char *cdrptr;
20          int cdrplen;
21      };

22      struct  {                     /* CDR for vector  2 3 4  */
23          struct cdrdesc desc;
24          int  rho;
25          int  data[3];
26      } v234 = {CDRID+16,3,'I',4,1,3,2,3,4},
27        *ptr_v234;                  /* Pointer to v234        */

28      struct {                      /* CDR for  matrix result */
29          struct cdrdesc desc;
30          int  rows;
31          int  cols;
32          struct cdrptr pointers;
33      } *result;

34      char *result_data;           /* used to point to result data */

35      int token=0,i,j;
36      union {
37          int code;                /* Return code as 1 fullword  */
38          short rc[2];   /* Return code as 2 halfwords */
39      } rc;
40      char parms[]="SM(OFF) WS(200K)"; /* APL initialization parms */
41      int len=sizeof(parms)-1;     /* Length of parms          */
```

Figure 30 (Part 1 of 2). C Program Demonstrating Use of the APLP and APLF Requests

```
    /*  ---- Call APL2PI to initialize APL2 ----  */
42      APL2PI("INIT",&token,&rc.code,"SAMPLE ",0,0,0,len,parms);
43      if(rc.code > 1) goto error1;

    /*  ---- Call APL2PI to enter STATS namescope ---- */
44      APL2PI("APLP",&token,&rc.code,"PKGLIB.STATS ");
45      if(rc.code != 0) goto error2;

    /*  ---- Call APL2PI to execute ⎕NL 2 3 4 ---- */
46      result = 0;
47      ptr_v234 = &v234;
48      APL2PI("APLF",&token,&rc.code," ","⎕NL ",&result,0,&ptr_v234);
49      if(rc.code != 0) goto error2;

    /*  ---- Display result returned by APL2 ---- */
50      printf("\nResult returned from execution of ⎕NL 2 3 4\n\n");
51      printf("%s%x\n%s%d\n%s%c\n%s%x\n%s%hd\n%s%i %i\n%s\n%s\n",
52         "CDRDLEN = ",result->desc.cdrhdr.cdrdlen,
53         "CDRXRHO = ",result->desc.cdrxrho,
54         "CDRRT   = ",result->desc.cdrrt,
55         "CDRRL   = ",result->desc.cdrrl,
56         "CDRRANK = ",result->desc.cdrrank,
57         "CDRRHO  = ",result->rows,result->cols,
58         "CDRDATA:",
59         "--------");

60      result_data=result->pointers.cdrptr;

61      for (i=0;i<result->rows;i++){
62          for (j=0;j<result->cols;j++)
63              putchar(*result_data++);
64          putchar('\n');
65      }

66      printf("--------\n\n");

    /*  ---- Call APL2PI to exit STATS namescope ---- */
67      APL2PI("APLP",&token,&rc.code);

    /*  ---- Call APL2PI to terminate APL2 ----  */
68     shutdown:
69      APL2PI("TERM",&token,&rc.code);
70      return 0;

71     error1:
72      printf("Unexpected error during APL2 initialization\n");
73      printf("Return code: %hd %hd\n",rc.rc[0],rc.rc[1]);
74      return;

75     error2:
76      printf("Unexpected error during APL2 function execution\n");
77      printf("Return code: %hd %hd\n",rc.rc[0],rc.rc[1]);
78      goto shutdown;

79  }
```

Figure 30 (Part 2 of 2). C Program Demonstrating Use of the APLP and APLF Requests

Notes

- Lines 3, 6-33: this routine makes use of the `'APLF'` call which requires that arguments and results passed to and from APL2 be provided in CDR format. CDR format is described in detail in Chapter 3, “APL2 Data Representation” on page 13.

  CDRs passed from C programs to APL2PI may be dense or pointer form CDRs; CDRs returned from APL2PI are always pointer form CDRs.

- Lines 6-15: the descriptor section of a CDR is defined as a C structure. Note that `cdrrho` is not included since this CDR field may be a null vector.

- Lines 16-21: the CDR pointer section is defined as a C structure.

- Lines 22-27: a dense form CDR representing the integer vector `2 3 4` is defined and initialized. The address of this CDR is assigned to ptr_v234 on line 47 and that address is passed as an argument on the `'APLF'` call on line 48.

- Lines 28-33: the `'APLF'` call on line 48 should produce a character matrix result. This result will be returned as the address of a pointer form CDR which is mapped by this structure.

- Line 44: an `'APLP'` call is issued to cause the *PKGLIB.STATS* namescope to be entered; subsequent APL2PI calls will be executed in that namescope. This technique is necessary if subsequent `'APLF'` calls request execution of primitive functions since the namespace argument cannot be provided on such calls.

- Line 48: an `'APLF'` call is issued to request execution of the system function ⎕*NL* with right argument `2 3 4`. Note that a system function name or a primitive function symbol can be specified as the function to be executed on an `'APLF'` call. If APL2 symbols are imbedded in character literals in a C program they may not be displayed correctly in the listing.

  The `result`, `larg` and `rarg` arguments of the `'APLF'` call are expected to be fullword fields which contain the addresses of CDRs. Therefore pointers must be specified, using the C “`&`” operator, when these arguments are passed on the call. Since a left argument is not provided for this call, the `larg` field is coded as 0 in the argument list.

Using the APL2PI Interface from COBOL

Many of the functions available on the APL2PI interface can be used in a simple and straight forward fashion in COBOL programs. Since COBOL only provides very rudimentary support for pointers, however, the 'APLF' and 'APLV' calls cannot be used effectively. COBOL service routines are not supported.

The following example shows a COBOL program which makes use of the APL2PI interface to generate a set of random numbers and to compute their average. The program illustrates simple use of the 'INIT', 'TERM', 'APLE' and 'APLS' calls. Lines of the program are numbered on the left for reference in the notes after the figure. The IBM VS COBOL II Program Product (5668-958) Version 1, Release 3 was used to construct this example. Other COBOL compilers may or may not have similar capabilities.

```
 1     Identification division.
 2       Program-id. callapl2.
 3     Environment division.
 4     Configuration section.
 5       Source-computer.  IBM-370.
 6       Object-computer.  IBM-370.
 7     Input-output section.

 8     Data division.
 9     Working-storage section.

10       1 TOKEN picture s9(9) binary value zero.
11       1.
12        2 RCODE.
13          3 RCODE1 picture s9999 binary.
14          3 RCODE2 picture s9999 binary.
15        2 RETCODE redefines RCODE picture s9(9) binary.
16       1 ZEROV picture s9(9) binary value zero.

17       1 OPTIONS picture x(16) value "SM(OFF) WS(200K)".

18       1 QNA-ATR picture x(14) value "0 11 ⎕NA 'ATR'".
19       1 GET-RANDOM picture x(20) value "'E8 1 *' ATR 5 ? 100".

20       1 RESULT-LENGTH picture s9(9) binary.
21       1 RESULT-BUFFER.
22         2 RESULTS computational-2 occurs 5 times.
23       1 NUMBERS-ARRAY.
24         2 NUMBERS picture -ZZ9 display occurs 5 times.
25       1 ITEM picture s9(9) binary.
26       1 ITEMS picture s9(9) binary value 5.
27       1 AVERAGE computational-2.
28       1 DISPLAY-AVERAGE picture -ZZ9.999 display.
```

Figure 31 (Part 1 of 2). COBOL Program Demonstrating Use of APL2PI

```
29     Procedure division.

30         Call "APL2PI" using
31             by content "INIT" by reference TOKEN RCODE
32             by content "SAMPLE " ZEROV ZEROV ZEROV
33             length of OPTIONS OPTIONS
34         If RETCODE is > 1 go to ERROR1
35         End-if

36         Call "APL2PI" using
37             by content "APLE" by reference TOKEN RCODE
38             by content length of QNA-ATR QNA-ATR
39             by content ZEROV " "
40         If RETCODE is not = 0 go to ERROR2
41         End-if

42         Move length of RESULT-BUFFER to RESULT-LENGTH
43         Call "APL2PI" using
44             by content "APLE" by reference TOKEN RCODE
45             by content length of GET-RANDOM GET-RANDOM
46             by reference RESULT-LENGTH RESULT-BUFFER
47         If RETCODE is not = 0 go to ERROR2
48         End-if

49         Perform with test after
50             varying ITEM from 1 by 1
51             until ITEM = ITEMS
52           move RESULTS(ITEM) to NUMBERS(ITEM)
53         End-perform
54         Display "Random numbers returned by APL2: "
55                 NUMBERS-ARRAY upon console

56         Call "APL2PI" using
57             by content "APLS" by reference TOKEN RCODE
58             by content "PKGLIB.STATS " "AVG " ZEROV " "
59             by reference ITEMS RESULT-BUFFER AVERAGE
60         If RETCODE is not = 0 go to ERROR2
61         End-if

62         Move AVERAGE to DISPLAY-AVERAGE
63         Display "The average is: " DISPLAY-AVERAGE upon console.

64     Shutdown.
65         Call "APL2PI" using
66             by content "TERM" by reference TOKEN RCODE
67         Stop run.

68     Error1.
69         Display "Error during APL2 initialization: "
70             RCODE1 " " RCODE2 upon console
71         Stop run.
72     Error2.
73         Display "Error during APL2 execution: "
74             RCODE1 " " RCODE2 upon console
75         Go to shutdown.
```

Figure 31 (Part 2 of 2). COBOL Program Demonstrating Use of APL2PI

Notes

- Lines 12-15: the return code returned by APL2PI is formally a pair of halfwords in a fullword field. In some situations it is useful to treat it as a single fullword; in others, as a pair of halfwords.
- Line 16: numeric literals cannot be specified as arguments in a COBOL CALL statement, and therefore must be given names in the data division.
- Lines 18-19: specify APL2 expressions that will later be executed by means of `'APLE'` calls to APL2PI. Note that APL2 characters can be specified in such expressions but may not print correctly in the COBOL program listing.
- Lines 30-33: APL2 is initialized by means of an `'INIT'` call to APL2PI. Note that arguments that are to be updated on the call must be passed by reference, while constants and arguments which are not expected to be updated are passed by content. If a “by content” argument is updated as a result of the call to APL2PI, the updated value will not be available in the COBOL program.
- Line 34: this program is set up to allow it to invoke APL2 or to be invoked by APL2. This is done by accepting a return code of either 0 or 1 from the `'INIT'` call.
- Line 39: in this particular example, the calling COBOL program will not bother to check the results of the *⎕NA* executed in this call, since the subsequent call will fail with a predictable error if the *⎕NA* fails. Therefore, the `RLENGTH` and `RESULT` fields are specified as `ZERO` and `" "` respectively. APL2PI will update the `RLENGTH` field with the length of the actual result, but that updated value will not be returned to the COBOL program because the `ZERO` argument was passed by content.

APL2 and COBOL Data Representations

APL2 typically represents numeric data in a number of different formats in the workspace. Real numbers are represented as double precision floating point values, integers are typically represented using fullword integer representation and Boolean values are often represented as bits. The representation of the value of a variable or the result of an expression is dependent upon the operations performed upon it, and cannot be simply predicted. Most of this is transparent to the APL2 user who sees numbers as numbers an lets the computer manage their representation in its internal memory.

COBOL, on the other hand, is a language in which data representation is visible to and carefully managed by the programmer. When data is passed between a COBOL application and APL2 using the APL2PI interface that data must be transformed to a representation acceptable to APL2 and/or the COBOL application. This same situation exists when other high level languages are used with the APL2PI interface. The APL2 external functions *PTA*, *ATP*, *ATR* and *RTA* are available to assist with such transformation. *PTA* and *ATP* are described in this document; *ATR* and *RTA* are described in the *APL2 Programming: Using the Supplied Routines* manual.

The following table shows the correspondence between types specified in the COBOL `USAGE` clause and those specified in the patterns used with the *PTA*, *ATP*, *ATR*, and *RTA* external functions:

COBOL Numeric type	Picture and USAGE	RT/RL in pattern
Binary	PIC S9999 BINARY	I2
	PIC S9(9) BINARY	I4
Internal Floating	COMPUTATIONAL-1	E4
	COMPUTATIONAL-2	E8
External Floating	PIC +9(3).99E+99 DISPLAY	none
External Decimal	PIC S9999 DISPLAY	Z5
Internal Decimal	PIC S9999 PACKED-DECIMAL	P3

In addition to allowing numeric data to be represented in these forms, COBOL also separately maintains a scale factor or decimal point position for binary and decimal representations (the scale factor is an inherent part of the floating point representation, and consequently does not have to be separately maintained). When COBOL computations are performed on binary and decimal data, COBOL aligns the data around the decimal point to achieve the desired results. When such data is passed to APL, the position of the decimal point is not passed. For example, if the variable

```
01 CASH PICTURE S9999.99 BINARY VALUE -1234.56.
```

was passed to APL, it would be received as the value `¯123456`. If the COBOL program treated the value as dollars and cents, and the APL2 program treated it as cents, no problem would exist. If the position of the decimal point was variable or significant, it would be lost unless passed as a separate explicit argument to APL.

Similarly, if the value `¯1234567` was placed by an APL application in the COBOL `CASH` field as defined above, it would be interpreted by COBOL as the value -1234.56. This is because only the data, and not the decimal position is passed between APL2 and COBOL.

This behavior becomes a little more complex when decimal (packed or external) data is passed from or to a COBOL program. On the System/370, packed and zoned decimal representations allow a very wide range of numbers to be represented (31 digits for packed and 15 for zoned). When a packed or zoned decimal number, passed from COBOL, is accessed with the *PTA* function, using the `'P'` or `'Z'` representation types, that number is converted into double precision floating point representation so that it can be subsequently processed by APL. This conversion may loose precision and may change an integral value to a non-integral one (i.e.: a real number). Worse, the *ATP* and *ATR* external functions will not accept floating point right arguments when a representation type of `'P'` or `'Z'` is specified in the pattern specified in the left argument. This problem can be circumvented by converting the data to fullword integers using the APL2 floor (`⌊`) primitive, namely,

```
   DATA←1.23×100
   DATA
123
   'P2 0' RTA 'P2 0' ATR DATA
DOMAIN ERROR
   'P2 0' RTA 'P2 0' ATR DATA
                     ^
   'P2 0' RTA 'P2 0' ATR ⌊DATA
123
```

An alternate and often preferable way to avoid such problems is to use the BINARY, rather than DISPLAY or PACKED-DECIMAL, usage clause in COBOL programs for integer data that is passed to or from APL.

Using the APL2PI Interface from PL/I

Most functions available on the APL2PI interface can be used in PL/I programs. The 'APLF' and 'APLV' calls and some variants of the 'APLX' call involve the use of data in CDR format, and are more difficult, but not impossible to handle in the PL/I environment. PL/I service routines are not supported.

The following example shows a simple PL/I program which makes use of the APL2PI interface to call the function *AVG* in namespace *PKGLIB.STATS* to obtain the average of a vector of numbers passed to it. Lines of the PL/I program are numbered on the left for reference in the following notes. The IBM PL/I Optimizing Compiler Version 2, Release 2 (5668-909) was used to construct this example. Other PL/I compilers may or may not have similar capabilities.

Notes

- Line 3: the APL2PI entry point must be declared in PL/I programs as shown on line 3. This declaration ensures that APL2PI will be called with the correct linkage conventions.
- Line 6: the return code returned by APL2PI is formally a pair of halfwords in a fullword field. In some situations it is useful to treat it as a single fullword; in others, as a pair of halfwords.
- Line 9: numeric literals cannot be specified as arguments in a PL/I CALL statement, and therefore must be given names by means of declarative statements.
- Line 14: this program is set up to allow it to invoke APL2 or to be invoked by APL2. This is done by accepting a return code of either 0 or 1 from the 'INIT' call.
- Line 20: note that APL2PI always updates the LEN field as the result of an 'APLS' call, thereby destroying the initial value of this field. If a subsequent 'APLF' call was made by this program, the LEN field would have to be reset before the call.

```
 1  *PROCESS OPT(2);
 2   PLI2APL: proc options(main reentrant) reorder;

 3   dcl APL2PI entry options(asm inter);
 4   dcl (NUMBERS(100), RESULT) float bin(53);
 5   dcl (TOKEN init(0), RC, SIZE, LEN init(0)) fixed bin(31);
 6   dcl RETCODE(2) fixed bin(15) based(addr(rc));
 7   dcl PICSIZE pic 'Z9';
 8   dcl (PICRC1, PICRC2) pic 'ZZZ9';
 9   dcl ZERO init(0) fixed bin(31) static;
10   dcl OPTIONS char(16) init('SM(OFF) WS(200K)') static;
11   dcl OPTLEN init(16) fixed bin(31) static;
12   dcl BUFFER char(72);

      /* ---- call APL2PI to initialize APL2 ---- */
13   call APL2PI('INIT',TOKEN,RC,'SAMPLE ',ZERO,ZERO,ZERO,OPTLEN,OPTIONS);
14   if RC > 1 then goto ERROR1;

15   display('Enter number of numbers to average ') reply(BUFFER);
16   SIZE = BUFFER;
17   PICSIZE = SIZE;
18   display('Enter ' || PICSIZE || ' numbers ') reply(BUFFER);
19   get string(BUFFER) list((NUMBERS(I) do I=1 to SIZE));

     /* ---- call APL2PI to compute average ---- */
20   call APL2PI('APLS',TOKEN,RC,'PKGLIB.STATS ','AVG ',LEN,' ',
                 SIZE,NUMBERS,RESULT);
21   if RC ¬= 0 then goto ERROR2;

22   display('The average is: '||RESULT);

     /* ---- call APL2PI to terminate APL2 ---- */
23   SHUTDOWN:
24   call APL2PI('TERM',TOKEN,RC);
25   return;

     /* ---- unexpected error during APL2 initialization ---- */
26   ERROR1:
27   display('Unexpected error during APL2 initialization');
28   PICRC1 = RETCODE(1);
29   PICRC2 = RETCODE(2);
30   display('Return Code:' || PICRC1 || PICRC2);
31   return;

     /* ---- unexpected error during APL2 function execution ---- */
32   ERROR2:
33   display('Unexpected error during APL2 function execution');
34   PICRC1 = RETCODE(1);
35   PICRC2 = RETCODE(2);
36   display('Return Code:' || PICRC1 || PICRC2);
37   goto SHUTDOWN;

38   end; /* PLI2APL */
```

Figure 32. PL/I Program Demonstrating Use of the APLF Request

Chapter 21. Implementation Details

The APL2PI interface consists of a complex set of routines which can be used in a variety of ways to allow APL2 and non-APL applications to interact. For many applications the documentation provided in Chapter 16 thru Chapter 20 will be entirely sufficient. In more sophisticated applications, however, it may be useful to understand the interplay of the routines.

The APL2PI interface routines can be divided into two major groups:

1. The routines which provide interfaces used by the non-APL application. The main routine in this group is `APL2PI`. `APL2PI` is the entry point to which control is passed from the non-APL application when any request is made to the interface.
2. A group of APL2 external functions providing the interfaces used by APL2 routines or the APL user when communicating with the non-APL application. The two important functions in this set are *APL2PI* and *APL2PIE*. Note that this *APL2PI* external function is not the same as the APL2PI routine used by non-APL application programs. The *APL2PI* external function is simply a niladic cover function for the *APL2PIE* external function. Its use will be described in more detail below.

The non-APL application which uses the APL2PI interface can be invoked independently or from an active APL2 session. If invoked independently, the non-APL application causes APL2 invocation to occur on the first call to the APL2PI interface (typically an `'INIT'` call). To invoke a non-APL application from an active APL session, the APL2 user makes use of the *APL2PIE* external function. This external function activates the APL2PI interface and through it causes the non-APL application to be invoked. Once the non-APL application is so initialized, it can call the already active APL2PI interface to make requests to the pendant APL2 session.

It is possible for both modes of operation to be used together. For example, a non-APL application could be invoked using appropriate CMS or TSO commands, and that application could use `APL2PI` to cause APL2 to be invoked and to submit requests to it. One or more of those requests could cause one or more non-APL applications to be activated from the APL2 environment. All of these non-APL applications could interact using the facilities provided with the `APL2PI` module and the *APL2PIE* external function. Note however, that at any given time only one application (APL or non-APL) is running—all of the other applications are in a pendent state.

Non-APL applications invoked independently or from an active APL environment are often mainline programs written in a compiled high level language. Invocation of such compiled language mainline programs typically cause a programming environment to be established. Care must be taken when more than one non-APL application program is activated. Certain languages or versions of languages do not support more than one instance of the programming environment at any given time. For example, if one non-APL application written in COBOL is activated, it may not be possible to activate a second one written in the same language, because the second instance of its programming environment might interact destructively with the first instance.

Invoking APL2 from a Non-APL Application

When a non-APL application, invoked and running independently of APL, wishes to access APL2 facilities it does so by calling the APL2PI entry point. This entry point is in the AP2VAPI object module for VM/CMS or the AP2TAPI object module for MVS/TSO. That module can be link edited with the non-APL application, or it can be dynamically loaded or located by the non-APL application (by using the LOAD macro, for example, or the CMS NUCXLOAD macro). Once the APL2PI routine is available, the non-APL application passes requests to it by using the standard OS CALL protocol described in Chapter 17.

The first call typically issued by the non-APL application is an 'INIT' call to request initialization of APL2. If 'INIT' is not the first call made, the APL2PI interface recognizes that APL2 has not yet been invoked and automatically issues the equivalent of an 'INIT' call with default APL2 initialization parameters. Whether this 'INIT' call is issued implicitly or explicitly, APL2PI then causes APL2 to be started much as if it had been invoked from a CMS or TSO command line.

Note: The CMS and TSO command services are not actually used. Among other things, this means that a real APL2 command name must be used, and not the name of an EXEC or CLIST that would invoke APL2.

The normal default command used to invoke APL2 will contain the invocation options QUIET and RUN(APL2PI).

- The QUIET option suppresses display of the APL2 greeting messages and the interaction associated with establishing a connection to the *APL2PI* function. Its use is not essential to the proper operation of the APL2PI interface, but it is recommended since this output tends to be confusing when running an application. QUIET will be turned off once the interface is initialized, so that APL2 output during execution will be displayed normally.
- The RUN(APL2PI) option is required for proper operation of the interface. It causes the *APL2PIE* external function to be invoked (indirectly, with argument 0).

The *APL2PIE* external function then becomes a “co-routine” with the APL2PI routine used by the non-APL program; that is, each call from one to the other is treated as a return by the other one. *APL2PIE* begins by calling APL2PI, which treats this as a return from its request to start APL2.

At this point, APL2PI recognizes that the invocation of APL2 is complete and passes control back to the non-APL application if an explicit 'INIT' call was issued, or proceeds with the non-APL application's request if the 'INIT' call was implicitly issued.

When and if the non-APL application issues a request other than 'INIT' or 'TERM' to the APL2PI interface, APL2 will be in a state where it is awaiting the completion of its call to the *APL2PIE* external routine. 'APLE', 'APLS', 'APLF' and 'APLV' requests are completed by APL2PI using “callback” requests to APL2, i.e. a combination of 'XE' and 'XF' service calls as documented in “XE: Evaluate an APL Expression” on page 106 and “XF: Form or Find an APL Object” on page 109 of Chapter 14.

The 'APLP' request is executed by passing control from APL2PI to *APL2PIE* which executes the request using local logic and then passes control back to APL2PI.

The 'APLX' request causes APL2PI to pass control to *APL2PIE* which returns control to the APL2 session or application that called it. (See -- Heading 'APLXIC' unknown -- for information about the message which is displayed or returned in this situation.) When the user or APL application subsequently issues an *APL2PI* or *APL2PIE* 0 '' request, control is passed back to the APL2PI routine. APL2PI sets return codes and return values appropriately and returns control to the non-APL application which called it.

When the non-APL application issues a 'TERM' request, APL2PI calls *APL2PIE* which terminates the APL2 session by triggering)*OFF* command processing. APL2PI then itself terminates, returning control to the non-APL application.

Invoking a Non-APL Application from APL2

A non-APL application can be invoked from the APL2 environment using the following call to *APL2PIE*:

```
arguments APL2PIE 1 'name'
```

When *APL2PIE* receives this request, it calls the external routine APL2PI with a request to activate the specified non-APL application routine. APL2PI initializes the interface and then loads and calls the non-APL application routine.

In the CMS environment, APL2PI first looks for an existing CMS nucleus extension whose name matches that of the specified routine. If one is found, its address is used as the entry point address for the application. If no matching CMS nucleus extension is found, APL2PI issues a CMS NUCXLOAD for a relocatable CMS module with the specified name. In either case, APL2PI then calls the application using CMS SVC 202 conventions.

In the TSO environment, APL2PI issues an MVS LOAD for the specified routine, and calls it as a TSO command processor.

When the non-APL application receives control, it can issue calls to APL2PI to make service requests. The first request issued should be an 'INIT' request. Although APL2 and APL2PI have already been initialized, this 'INIT' request allows the non-APL application to identify itself by name to the APL2PI interface, and allows a service routine to be specified if desired.

Other APL2PI requests can be subsequently issued by the non-APL application. APL2PI processes these requests by invoking APL2 services or resuming APL programs, and eventually returning control to the non-APL application. See “Invoking APL2 from a Non-APL Application” on page 173 for further details.

Before terminating, the non-APL application should issue a 'TERM' request to APL2PI. On return from this service call, the application is then free to terminate. When it does so, control is returned to APL2PI, since it originally invoked the non-APL application. APL2PI in turn returns control to the *APL2PIE* external function which returns control to the APL2 session or to the APL function that last called it.

Environment Isolation

When APL2 is invoked, it establishes exits so that asynchronous or unexpected events (attention signals, program checks, ABENDs, etc.) are captured and properly handled. Many non-APL applications, particularly those written in high level languages, need to do much the same thing. When mediating between APL2 and non-APL applications, APL2PI must take care to keep the APL2 and non-APL environments separate so that exits do not interfere with each other, and so that each application is properly notified of events appropriate to it.

In the TSO environment, this is done by using MVS task isolation. APL2 and each non-APL application is established as a separate MVS task. APL2PI activates and deactivates the appropriate tasks (using WAIT and POST) as control flows between APL2 and a non-APL application. Since each MVS task may have its own set of exits, no conflicts exist between APL2 and any of the non-APL applications and events are directed to the task that is currently active.

In the CMS environment, no equivalent level of task isolation is available. Each time control crosses between the non-APL and APL2 applications, APL2PI saves the SPIE, STAE, and STAX information for the application giving up control and reestablishes the corresponding information for the application to which control is being passed.

Note: APL2PI makes no attempt to control or detect CMS ABNEXIT or CMS simulation of DOS/VSE STXIT. This should not cause any problem unless

- a non-APL application requests an ABEND using DMSABN, or
- a non-APL application activates the CMS DOS environment.

Neither of these actions is supported for programs interacting with APL2.

There is another aspect of the CMS environment that can cause problems. When a command is issued by an application program using SVC 202, a level is added to the CMS SVC save area chain. APL2 is dependent on running all of its operations at the same CMS SVC level, and would ABEND if it passed control to an external routine which changed the SVC level and returned to APL2. For this reason APL2PI simulates an SVC 202 linkage rather than simply issuing an SVC 202 to invoke a non-APL application. In this way the non-APL application and APL2 both operate at the same CMS SVC level as they pass control back and forth.

There are many cases in CMS where applications can issue commands and thereby cause additional levels to be added to the SVC chain. For example, a REXX application could call an XEDIT session which in turn could call an XEDIT macro which could execute CMS commands. Each of these calls would introduce another level to the SVC chain. If such an application were to make calls to APL2PI, all calls would have to occur at the same SVC level. If that application was invoked from APL2 via *APL2PIE*, then all calls to APL2PI would have to be made from the SVC level at which the application was invoked by APL2.

Appendixes

Appendix A. Service Parameter Summary

In the following table:

- **boldface** parameters are values you must supply.
- **UPPERCASE** parameters are keywords you must specify.
- *italic* parameters have values returned by the system.
- ***underlined italic*** parameters have values you must supply, but which will be replaced by the system.

Figure 33 (Page 1 of 2). Service Parameter Summary

	Parm2	Parm3	Parm4	Parm5	Parm6	Parm7	Parm8	Parm9
Chapter 6, Code D__: Data Conversion Services								
DE	*retcode*	**datalen**	*outbuff*	**data**				
DN	*retcode*	**bufflen**	*outbuff*	**data**	**types**	**index**	**count**	[**descrip**]
DU	*retcode*	**datalen**	*outbuff*	**data**	**trantab**			
DX	*retcode*	***outlen***	*outbuff*	**data**	**datalen**	**options**	[**charset**]	
DZ	*retcode*	**datalen**	*outbuff*	**data**				
Chapter 7, Code E__: Error Handling Services								
ED	**dumpid**	[**psw**	[**regs**]]					
ET	**abcode**							
EX	**exitl0**							
EZ	*retcode*	**entryadd**	**token**					
Chapter 8, Code F__: File System Services								
FA	*retcode*	**libno**	**fname**	**pass**	*token*	**usetype**	[*maxlen*	[*records*]]
FC	*retcode*	**libno**	**fname**	**pass**	**maxsize**	**usetype**	[**maxlen**]	
FD	*retcode*	**libno**	**fname**	**pass**				
FG	*retcode*	**OS**	***token***	**gname**	**fname**			
FG	*retcode*	**SR**	***token***	**bufflen**	*buffer*	***reclen***		
FG	*retcode*	**CL**	***token***					
FL	*retcode*	**libno**	**pass**	**bufflen**	*buffer*	[*start*	[*end*]]	
FR	*retcode*	**bufflen**	*buffer*	***recno***	***token***			
FS	*retcode*	**libno**	**fname**	**pass**	**maxsize**			
FW	*retcode*		**buffer**	***recno***	***token***			
FZ	*retcode*				***token***			
Chapter 9, Code M__: Message Services								
MC	*retcode*	**msgnum**						
MF	*retcode*	**msgnum**	**opt**	[**fill-val**	**fill-len**]	**. . .**		
MF	*retcode*	*msgnum*	*opt*	*outarea*	***outlen***	[**fill-val**	**fill-len**]	**. . .**
Chapter 10, Code P__: Process Services								
PP	*ecb*	**postcode**						
PT	***ecb***	**time**						
PW	[*posted*	***ecb***	[***ecb***]	**. . .**]				
Chapter 11, Code SC: Shared Variable Services								
SC	***svpblock***							
Chapter 12, Code T__: Terminal Services								
TA	*action*	***resource***	***signal***					
TZ	**action**							

Figure 33 (Page 2 of 2). Service Parameter Summary

	Parm2	Parm3	Parm4	Parm5	Parm6	Parm7	Parm8	Parm9

Chapter 13, Code V__: Virtual Storage Services

VF	**length**	**address**
VG	**length**	*address*
VP	**length**	*address*
VQ	**length**	**address**
VV	***length***	*address*
VX	***length***	*address*

Chapter 14, Code X__: External Call Services

XB	***ecv***	
XC	**token**	*area*
XD	**token**	*area*
XE	***ecv***	
XF	***ecv***	
XG	***ecv***	

Appendix B. Testing and Using Processors

Complete information on making user-written processors part of the APL2 system is provided in *APL2 Installation and Customization under CMS* and *APL2 Installation and Customization under TSO*. This appendix outlines only one approach for each case. A number of variations are possible.

Accessing a Local Auxiliary Processor under CMS

1. Compile the processor, creating a CMS **TEXT** file.
2. Use the **AP2MV2AP EXEC** supplied with APL2 to put the processor into a **LOADLIB**. If you want to use more than one local AP in this way they must all be put into the same **LOADLIB**. For example, if your AP is named OUR517:

   EXEC AP2MV2AP OUR517 OUR517 TEXT * MYAPS LOADLIB A
3. Use the CMS **FILEDEF** to associate the **LOADLIB** with name **AP2LOAD**. (This is often done in an **AP2EXIT EXEC**.) For example:

   FILEDEF AP2LOAD DISK MYAPS LOADLIB *
4. Include the **APNAMES** parameter in your APL2 invocation. For example:

   APL2 APN(OUR517)
5. Use ⎕*SVO* to share variables with the AP. For example:

   ```
   517 ⎕SVO 'VAR'
   ```

Accessing an External Routine under CMS

1. Compile the routine, creating a CMS **TEXT** file.
2. Use the CMS **LKED** command to place the module into a **LOADLIB**. You can access as many loadlibs as you wish, so you may find it simpler to use a separate one for each external routine. For example, if your external routine is named PARSE:

   FILEDEF SYSLMOD CLEAR
   FILEDEF SYSLMOD DISK PARSE LOADLIB A (RECFM U
   LKED PARSE(RENT
3. Create a **NAMES** file defining the syntax of your routine. Names files are described in *APL2 Programming: System Services Reference*. As an example, you might place this in a file named **PARSE NAMES**:

   ```
   :NICK.PARSE :LOAD.PARSE :MEMB.PARSE :LINK.FUNCTION :LARG. :RARG.
   ```
4. Within APL2, use ⎕*NA* to associate a name in your workspace with the external routine. For example:

   ```
   '(PARSE)' 11 ⎕NA 'PARSE'
   ```

Accessing a Local Auxiliary Processor under TSO

1. Compile the processor, creating a TSO **OBJ** file.
2. Use the TSO **LINK** command to convert the object file to a load module and place it in a load library. You may want to keep all of your private load modules in one library. For example, if your AP is named OUR517:

   LINK OUR517 LOAD(PRIVATE(OUR517)) AMODE(31) RENT
3. Allocate filename **LOADLIB** to your load module library before invoking APL2. For example:

   ALLOC F(LOADLIB) DA(PRIVATE.LOAD)
4. Include the **APNAMES** parameter in your APL2 invocation. For example:

   APL2 APN(OUR517)
5. Use ⎕*SVO* to share variables with the AP. For example:

   ```
   517 ⎕SVO 'VAR'
   ```

Accessing an External Routine under TSO

1. Compile the processor, creating a TSO **OBJ** file.
2. Use the TSO **LINK** command to convert the object file to a load module and place it in a load library. You may want to keep all of your private load modules in one library. For example, if your external routine is named PARSE:

   LINK PARSE LOAD(PRIVATE(PARSE)) AMODE(31) RENT
3. Create a **NAMES** file member defining the syntax of your routine. Names files are described in *APL2 Programming: System Services Reference*. You may want to keep all of your private names files in one library. As an example, you might place this in a member named **PARSE**: of a partitioned data set named **MY.NAMES**.

   ```
   :NICK.PARSE :LOAD.MYFUNS :MEMB.PARSE :LINK.FUNCTION :LARG. :RARG.
   ```
4. Allocate your load module library using the name specified in the **NAMES** file. For example:

   ALLOC F(MYFUNS) DA(PRIVATE.LOAD)
5. Allocate your names file library. For example:

   ALLOC F(MYNAMES) DA(MY.NAMES)
6. Within APL2, use ⎕*NA* to associate a name in your workspace with the external routine. For example:

   ```
   '(MYNAMES)' 11 ⎕NA 'PARSE'
   ```

Appendix C. Macros Intended for Customer Use

The macros identified in this appendix are provided as programming interfaces for customers by APL2 Version 2. These macros are written in assembler language. They can be used as models for writing macros for other languages if desired.

Warning: Do not use as programming interfaces any APL2 Version 2 macros other than those identified in this appendix.

The following set of macros is for Auxiliary Processors written to the APL2 interfaces, and for Associated Processor 11 routines:

Mapping Macro	MVS Distribution Library	CMS Macro Library
AP2CDR	APL2.AAP2MACS	AP2MAC MACLIB
AP2CMDC	APL2.AAP2MACS	AP2MAC MACLIB
AP2CSVPE	APL2.AAP2MACS	AP2MAC MACLIB
AP2ECV	APL2.AAP2MACS	AP2MAC MACLIB
AP2PCV	APL2.AAP2MACS	AP2MAC MACLIB
AP2PTH	APL2.AAP2MACS	AP2MAC MACLIB
AP2RC	APL2.AAP2MACS	AP2MAC MACLIB
AP2SCV	APL2.AAP2MACS	AP2MAC MACLIB
AP2SDF	APL2.AAP2MACS	AP2MAC MACLIB
AP2TCMS	APL2.AAP2MACS	AP2MAC MACLIB
AP2XPTX	APL2.AAP2MACS	AP2MAC MACLIB

If you need to use the mapping macros, ask your system programmer:

- Under CMS, you need to know what disk APL2 is installed on.
- Under MVS, you need to know the dataset name of the distribution library for the APL2 macros.

The following set of macros is for Auxiliary Processors written to VS APL interfaces.

Mapping Macro	MVS Distribution Library	CMS Macro Library
APLDESC	APL2.AAP2MACS	AP2VSAPL MACLIB
APLKSTOZ	APL2.AAP2MACS	AP2VSAPL MACLIB
APLKZTOS	APL2.AAP2MACS	AP2VSAPL MACLIB
APLPCV	APL2.AAP2MACS	AP2VSAPL MACLIB
APLREGS	APL2.AAP2MACS	AP2VSAPL MACLIB
APLSCVS	APL2.AAP2MACS	AP2VSAPL MACLIB
APLSHSVP	APL2.AAP2MACS	AP2VSAPL MACLIB
APLZCODE	APL2.AAP2MACS	AP2VSAPL MACLIB
AP2CDR	APL2.AAP2MACS	AP2VSAPL MACLIB
AP2SDF	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVDFORM	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPACC	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPCPY	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPOFR	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPQRY	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPREF	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPRET	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPSOF	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPSON	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPSPC	APL2.AAP2MACS	AP2VSAPL MACLIB
ASVPWAIT	APL2.AAP2MACS	AP2VSAPL MACLIB

If you need to use the mapping macros, ask your system programmer:

- Under CMS, you need to know what disk APL2 is installed on.
- Under MVS, you need to know the dataset name of the distribution library for the APL2 macros.

Index

Special Characters

A

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W

We'd Like to Hear from You

APL2 Programming:
Processor Interface Reference
Version 2 Release 1

Publication No. SH21-1058-00

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: (408) 463-4488.
- Electronic mail—Use one of the following network IDs:
  - IBMMail: USIB6JN8
  - Internet: apl2@vnet.ibm.com

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

APL2 Programming:
Processor Interface Reference
Version 2 Release 1

Publication No. SH21-1058-00

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

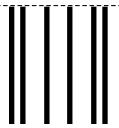
Company or Organization

Phone No.

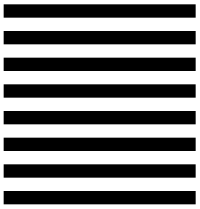
Readers' Comments
SH21-1058-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department M46/D12
PO Box 49023
San Jose, CA 95161-9023

Fold and Tape **Please do not staple** Fold and Tape

SH21-1058-00

Cut or Fold
Along Line

File Number: S370-34
Program Number: 5688-228

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SH21-1058-00

Spine information:

APL2 Programming: Processor Interface Reference *Version 2 Release 1*